



PHD-FSTC-2017-57

THE FACULTY OF SCIENCE, TECHNOLOGY AND COMMUNICATION

DISSERTATION

Presented on the 16th October 2017 in Luxembourg

to obtain the degree of

**DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE**

by

LOÏC GAMMAITONI

Born on the 12th December 1990 in Thionville (Moselle, France)

On the Use of Alloy in Engineering Domain Specific Modeling Languages

Dissertation Defense Committee

Dr. ULRICH SORGER, Chairman

Professor, University of Luxembourg

Dr. PIERRE KELSEN, Dissertation Supervisor

Professor, University of Luxembourg

Dr. BENOIT COMBEMALE, Jury Member

Professor, University of Toulouse (France)

Dr. MANUEL ALCINO CUNHA, Jury Member

Assistant Professor, University of Minho (Portugal)

Dr. NICOLAS NAVET, Jury Member

Assistant Professor, University of Luxembourg

Abstract

Domain Specific Modeling Languages (DSMLs) tend to play a central role in modern design processes as they enable the effective involvement of domain experts by focusing on a particular problem domain while abstracting away technical details. In this thesis, we investigate the specification of DSMLs with a particular focus on domain expert driven validation. Mainly, we are interested in developing Alloy-based approaches, allowing the definition of specifications from which instances can be generated and given to the domain experts for the sake of validation.

The work we present in this thesis can be divided into three parts:

The first part concerns the definition and execution of model transformations defined in Alloy. While Alloy analysis can be used as an execution engine for model transformations, the analysis process is time consuming. Model transformations playing a central role in DSML definitions, the development of a new model transformation language, named F-Alloy, retaining the benefits of Alloy with the added property of being efficiently computable was necessary.

The second part focuses on validation. In that domain, our first contribution is a novel approach to the validation of model transformations called Visualization-Based Validation (VBV). VBV relies on the review by domain experts of intuitive depictions of model transformation traces to validate model transformation specifications. The whole process is made efficient by the usage of hybrid analysis, a combination of Alloy analysis and F-Alloy interpretation, allowing to reduce the time needed to analyze model transformations to the time needed to analyze its source. Our second contribution in the validation area is the definition of an Alloy-based approach to the specification and validation of DSMLs and of a design process defining how DSMLs can be validated using Alloy analysis at each iteration of the process. More precisely, we present how the abstract syntax, concrete syntax and operational semantics of a DSML can be defined using Alloy and F-Alloy, and show that the validation of a DSML's abstract syntax and semantics benefits from the application of its concrete syntax.

The third and last part aims at bringing those contributions to the practical world. To achieve this we developed a tool named Lightning implementing the aforementioned contributions. This tool, which belongs to the category of language workbenches, has been successfully used in an inter-disciplinary collaboration to define the Robot Perception System Language (RPSL). Based on this definition of RPSL, a framework has been developed to allow the execution of so called design space explorations. This framework represents a successful application of our approach to the real world problem of having RPSL specifications validated by experts in robotics.



Acknowledgment

I do believe that writing a proper acknowledgment is exceedingly more difficult than to write any other part of this work. Indeed, while there is no page limit to synthesize my work of 4 years, only one page is to be allocated to the acknowledgment of all that has made me what I am today. Doomed to fail expressing due regards to all those who enriched me and thus this work indirectly, I resign myself to mention only the top of the iceberg, namely, those I think are the main contributors of my hypothetical success.

My deepest thanks go to my thesis supervisor, Pierre KELSEN, who successfully communicated to me his taste for fundamental research. There was not a single day where I was not looking forward to our next discussion. Thank you so much for giving me the opportunity to work with you, for ~~always~~ sometimes doubting me and nevertheless for always being there supporting me. I would also like to thank my team members Christian GLODT, Qin MA and Christophe KAMPHAUS for their support and interest in my work. Without you two, Christian and Christophe, Lightning wouldn't be as polished, and without you Qin, the theory and use of compound model transformations wouldn't have been as developed. A special thank goes to Nico HOCHGESCHWENDER, who not only allowed me to validate my work on his research problem, but also gave me the necessary confidence in the usefulness of my work. Amongst other researchers, I would like to thank Nicolas NAVET and Yves LE TRAON for relentlessly following the advances of my work since its beginning and for the constructive feedbacks provided. Thanks also to Alcino CUNHA and Benoit COMBEMALE for accepting the invitation to join my thesis defense committee and Ulrich SORGER for accepting chairing it. Without you, concluding this memorable chapter of my life wouldn't have been possible.

Investing so much time and effort on a single piece of work requires patience, passion and determination. Lacking patience, I can only thank my family and friends for helping me stay on the right track. More precisely, I would like to dedicate this entire thesis to my late grand-father Bruno ANDERLINI, who left us a year ago. Bruno was an upright man who always valued hard work and honesty. I wouldn't be who I am today without him and am thus forever beholden to him. Thanks also to his wife, my lovely "mami" Josianne and to Bruna and Alfredo GAMMAITONI. I could never imagine having more caring grandparents. Thanks to my adventurer of a godfather Alain ANDERLINI. You gave me your passion for travels and discovery as well as an open mind.

Thanks to my partner Xuejun LI, for having to deal with me on a daily basis. I know it is not an easy task, and thus cannot thank you enough for managing it so brilliantly. You are the best. 我爱你. Thanks also to Luca CIPPICIANI, my good friend of old, with whom I went through thick and thin and on which I can always rely.

Last but not least, I thank my sister Gaëlle for all the light she brings in my life and my parents, Cathy and Claude, for their everlasting love and support. Thank you dad for acting as a model for my sister and me all those years, and for showing me how a caring and fully devoted father should be. Mum, you've shown and keep showing me what unconditional love means ... and it means all to me. **I love you all, deeply.**

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Objectives	2
1.3	Contributions	3
1.4	Publications	5
1.5	Thesis Outline	6
2	Background	9
2.1	Model Driven Engineering	9
2.1.1	Models	9
2.1.2	Model Transformations	10
2.2	Software Language Engineering	12
2.2.1	Domain Specific Languages	12
2.2.2	Abstract Syntax	12
2.2.3	Concrete Syntax	12
2.2.4	Semantics	13
2.2.5	Language Workbenches	13
2.3	The Alloy Language	13
2.3.1	An Informal Alloy Introduction Based on Ecore	14
2.3.2	Alloy Modules and Instances	14
2.3.3	Signatures and Fields	15
2.3.4	Facts	16
2.3.5	Predicates and Functions	17
2.3.6	Assertions and Commands	17
2.4	Expressing Model Transformations in Alloy	18
2.4.1	A Generic Approach to the Expression of Model Transformations in Alloy	18
2.4.2	Related Work on the Use of Alloy in the Specification of Model Transformations	21
2.4.3	Benefits and Limitations of Alloy in the Specification of Model Transformations	21
3	The F-Alloy Language	23
3.1	From Alloy to F-Alloy	23
3.1.1	Motivation	23
3.1.2	A Mathematical Framework to Reason About Alloy Modules and Instances	25
3.1.3	Functional Alloy Modules	26
3.2	Case Studies	27

3.2.1	The CD and RDBMS Alloy Modules	27
3.2.2	CD2RDBMS: A Class Diagram to Relational Database Management System Model Transformation	29
3.2.3	CDRefinement: A Class Diagram Refinement Scenario	33
3.3	Syntax of F-Alloy	36
3.3.1	F-Alloy's Syntax	37
3.3.2	A Formal Definition of Mappings	37
3.3.3	F-Alloy's Syntax Usage	39
3.3.4	F-Alloy's Well-Formedness Constraints	41
3.3.5	F-Alloy to Alloy Correspondences	43
3.4	Translational Semantics of F-Alloy	44
3.4.1	Overview	44
3.4.2	Translating F-Alloy to Alloy	44
3.4.3	From F-modules to Functional Alloy Modules	51
3.5	Interpreting F-Alloy Specifications	58
3.5.1	Definition	58
3.5.2	Complexity Analysis	61
3.6	F-Alloy Interpretation Performance	61
3.6.1	F-Alloy vs Alloy	61
3.6.2	Comparing F-Alloy with Existing Model Transformations	62
3.6.3	Results	62
3.7	Related Work on Model Transformation Languages	63
3.8	Summary	64
4	On Model Transformation Validation	65
4.1	Validation of F-Alloy Specifications	65
4.2	Extending F-Alloy to Specify Compound Transformations	66
4.2.1	Syntactic Extension	66
4.2.2	Semantic Extension	67
4.3	Hybrid Analysis of (Compound) Model Transformations	69
4.3.1	Definition	69
4.3.2	Complexity Analysis	71
4.4	Visualization-Based Validation of Model Transformation	71
4.5	Application of VBV to CD2RDBMS	74
4.5.1	CD2RDBMS Specification and First F-Alloy Implementation	74
4.5.2	Definition of CD2RDBMS_viz	76
4.5.3	Example of VBV Iteration Using a Specific Test Case	77
4.5.4	Example of VBV Iteration Using Random Instance Generation	81
4.6	Evaluation of Hybrid Analysis	81
4.7	Related Work on Model Transformation Validation	82
4.8	Summary	84
5	An Approach Towards Defining DSMLs Using Alloy	85
5.1	The Structured Business Process Case Study	85
5.2	An Alloy-based Language Definition	86
5.2.1	Abstract Syntax	86
5.2.2	Concrete Syntax	88
5.2.3	Semantics	88
5.3	An Agile Design Cycle	89
5.3.1	Designing the Abstract Syntax	89

5.3.2	Designing the Concrete Syntax	90
5.3.3	Designing the Operational Semantics	92
5.4	Related Work on DSML Engineering Approaches	94
5.4.1	Alloy-based DSML Engineering Approaches	95
5.4.2	Example-Based Validation Approaches in the OMG World	95
5.4.3	On Concrete Syntax Definitions	96
5.5	Summary	97
6	The Lightning Language Workbench	99
6.1	Tool Presentation	99
6.1.1	Lightning Languages	99
6.1.2	Editor	100
6.1.3	Instance Viewer	100
6.1.4	Instance Editor	102
6.2	RPSL Meets Lightning, a Success Story	103
6.2.1	The RPSL Language	103
6.2.2	The Pick & Place Case Study	105
6.2.3	A Lightning-Based Design Space Exploration Framework	106
6.2.4	Alternatives Metamodel	110
6.2.5	A Design Space Exploration Scenario	110
6.3	Related Work on Language Workbenches	114
6.4	Summary	116
7	Conclusion	117
7.1	Contributions	117
7.2	Future Work	118
7.2.1	The F-Alloy Track	118
7.2.2	The Lightning Track	118
A	Functional Alloy Module Expressing the CD2RDBMS Transformation	121
B	Functional Alloy Module Expressing the CDRefinement Transformation	123
C	ATL Implementation of the CD2RDBMS Transformation	125
D	A Visual Language Model Expressed in Alloy: the LightningVLM	127
	Glossary	129
	Bibliography	133

TABLE OF CONTENTS

List of Figures

2.1	The OMG four-layer pyramid (taken from [1])	10
2.2	The big picture of model transformations (adaptation of [2] given in [3])	11
2.3	Example of a directory tree containing Alloy files	15
2.4	Headers of Alloy module a1, a2 and a3 corresponding to the directory tree given in fig.2.3	15
2.5	Alloy instance of the Alloy module given in Listing 2.1	16
2.6	Illustration of the approach to represent a model transformation in Alloy	19
3.1	Illustration of the usage of transformation functions w.r.t. the definition of functional Alloy modules	26
3.2	CD metamodel (adapted from [2])	28
3.3	RDBMS metamodel (adapted from [2])	28
3.4	Execution of the CD2RDBMS transformation	31
3.5	A simplified overview of the CD2RDBMS TGG transformation as given in [4]	31
3.6	TGG Rules composing the CD2RDBMS transformation proposed in [4])	31
3.7	Execution of the CDRefinement transformation	34
3.8	CDRefinement transformation defined using Henshin	34
3.9	An overview of the F-Alloy approach	36
3.10	Relation between Alloy and F-Alloy's semantics	44
3.11	A <code>CDRDBMS</code> -instance x with $f_{in}(x)$ highlighted in red and f_{out} highlighted in bold	52
3.12	A <code>CDRefinement</code> -instance, and the input and output of the transformation <code>CDRefinement</code> specifies as defined by f_{in} and f_{out} , respectively.	53
3.13	Time needed by the Alloy analyzer, the F-Alloy interpreter and the ATL engine to compute the <code>CD2RDBMS</code> transformation, given the same set of random CD-instances composed of 10 to 50 elements	62
3.14	Time needed by the Alloy analyzer, the F-Alloy interpreter and the Henshin interpreter to compute the <code>CDRefinement</code> transformation, given the same set of random CD-instances composed of 10 to 50 elements	63
4.1	Extending F-Alloy (from left to right) to support both basic and compound f-modules	66
4.2	Import Hierarchy of an f-module	71
4.3	Import Hierarchy of f_viz	72
4.4	Visualization-Based Validation (VBV) of the model transformation $f : m_{src} \rightarrow m_{dst}$	73
4.5	Import Hierarchy of <code>CD2RDBMS_viz</code>	76
4.6	Example of Class Diagram that can be given as input to the <code>CD2RDBMS</code> Transformation	77

4.7	Visualization of a bug in the computation of the CD2RDBMS transformation: FKKey missing for primary key columns obtained from associations.	79
4.8	Visualization of the case depicted in Fig.4.7 after refinement of the CD2RDBMS transformation.	79
4.9	Visualization of a bug in the computation of the CD2RDBMS transformation: non-primary attribute of non persistent classes are lost in the transformation	80
4.10	Visualization of the case depicted in Fig.4.9 after refinement of the CD2RDBMS transformation.	80
5.1	A Structured Business Process	85
5.2	Overview of an Approach to the Alloy-based Definition of Domain Specific Modeling Languages	87
5.3	Overview of our approach to the definition and rendering of a language's concrete syntax	87
5.4	Spiral diagram depicting how languages defined following the proposed approach can be incrementally designed	89
5.5	Raw visualization of a language model (using Alloy's Magic Layout)	90
5.6	Visualization of the instance depicted in fig. 5.5 using its concrete syntax definition	91
5.7	Faulty semantics execution of an SBP language model containing XORs	94
6.1	An overview of the Lightning Graphical User Interface	100
6.2	Outline view associated to the Alloy editor (currently editing the SBP Alloy module). Clicking an entry of this outline will have as effect to highlight where it has been declared in the editor.	101
6.3	An overview of the Lightning Tree-based instance editor	103
6.4	A youBot robot performing an insertion task at a service area.	104
6.5	Structural overview of an RPSL domain model and its conforming design alternative.	105
6.6	Visualization of a given Pick & Place configuration obtained with the framework. The feature model in which selected features are highlighted in green is at the top. At the bottom is a possible super graph for the given feature selection, with input and output ports displayed in green and red, respectively, and with components being either white or yellow depending on their associated weights.	107
6.7	An overview of the Lightning-based Framework designed to facilitate the design space exploration of the RPSL specification. Note that boxes represent Alloy modules, hexagones represent f-modules and the dashed arrows represent import relations between modules.	108
6.8	Overview of the usage of the Lightning-based framework in the performance of design space exploration. Here, a domain expert (bottom-left) can provide the framework with RPSL specifications, interact with the framework, and inspect graphically rendered design alternatives	113
6.9	Generic Language Workbench Feature Model proposed in [5] with highlighting of the features covered by Lightning	115

Chapter 1

Introduction

This introductory chapter aims at motivating and framing the work that has been accomplished during this thesis. We start by motivating our work in Section 1.1. In Section 1.2, we list the research objectives we aspire to fulfill. This aspiration led us to develop several contributions, listed in Section 1.3, leading to the publications listed in Section 1.4. The final section presents the outline of this thesis.

1.1 Motivation

Software engineering as the science of producing techniques aiming at easing the design, implementation and maintenance of software systems is very recent: only half a century old [6]. The complexity of software engineering problems is somehow proportional to the power of the machine they are meant to be run on [7], and the processing power of those machines grew exponentially in the past decades [8].

To cope with this overwhelming increase of complexity, software engineering techniques evolved quickly and a ramification of research fields ensued. This work places itself in the model driven engineering field, studying the use of *models* as means to provide solutions to some of the most recurrent problems in software engineering like optimization [9,10], *validation* and *verification* [11,12], consistency [13,14], reusability [15,16] and scalability [17,18] of software systems. Models are abstractions defining chosen aspects of a systems in terms of concepts, relations and constraints [19]. These abstractions provide an organized and sound way of designing systems and have various applications, from verification to code generation [20]. A recurrent challenge when it comes to defining a system is to handle the gap between the abstraction – i.e., the model – and the reality it defines – i.e., the system. Namely:

1. The customers for whom the system is developed have particular needs, proper to their domain of expertise, that they try to express in so-called requirements [21]. They are generally not familiar with the formalisms used to model their system and are hence unable to detect design errors before the latest stage, i.e., when they get hold of the system developed.
2. (Model) *engineers* are experts who model the system based on customers' requirements. Modeling is an error-prone exercise since by thinking abstractly of a given system, it is easy to overlook important details. Validation should thus be an integral part of a modeling process [22].

In this work we attempt to close this gap by combining the two following high-level approaches:

1. *Domain-specific modeling languages (DSMLs)* [23] are languages that can be used to present models in a form that customers (*domain experts*) are accustomed with.
2. *Alloy* [24] is a formalism in which systems can be modeled and that allows validating at any time models through automated *analysis*.

The motivation behind using DSML originates from the consensus [25, 26] that intuitive languages are required to enable effective communication between technical and non-technical experts. A plethora of work [27–29] show that DSML can be designed to let domain experts specify certain aspects of a system. From this popularity of DSMLs emerged new approaches to the verification and validation of DSML designs [30–32]. Yet those approaches (1) focus on specific aspects of a DSML definition (e.g., abstract syntax, or semantics) rather than considering a DSML as a whole and (2) are directed to modeling experts hence leaving a gap between the DSML specification and reality.

To tackle those drawbacks, we propose to build a language engineering framework atop of Alloy, a formal language used to specify and validate models ([33–36]), allowing the consideration of all components of a DSML specification to enable domain expert to validate them through the review of graphical visualizations and simulations of language models.

In this work, we choose to use Alloy as a main formalism to (1) focus on the essence of DSML, the language allowing us to make abstractions of platform specific concepts and (2) to use Alloy analysis as a foundation to our validation approaches.

Few works [37, 38] have investigated the use of Alloy in the specification of DSMLs, however, all limit themselves to the specification and validation of a single component of the DSML (generally abstract syntax and semantics alone).

In this thesis we investigate the use of Alloy in the specification and validation of DSMLs to provide a framework in which model engineers and domain experts can easily collaborate to validate model designs. This framework will allow the specification of all component of a DSML and will enable an efficient validation process relying on the usage of all the components specified.

A recent PhD thesis [39] also addresses the challenge of closing the gap between model engineers and domain experts in the specification of DSMLs. Yet, the focus of that thesis is on enabling the domain experts to provide partial specifications of the system to be modeled (through the automatic derivation of metamodels from sketches) rather than on the validation of specifications written by model engineers. This bottom-up approach to DSML specification requires multiple sketches as well as a model engineer fine tuning generated metamodels. Compared to this approach, we choose to leave the task of DSML specification to model engineers and rely on the domain experts for validation only. This choice allows us to escape challenges inherent to “modeling by demonstration” [40] (addressed in [39] by restraining the set of DSMLs that can be defined that way) while still actively involving domain experts in the DSML engineering processes.

1.2 Research Objectives

The thesis aims at addressing the common problem of model validation by allowing the involvement of domain experts at the earliest stage of design process. To do so, we focus on DSMLs specifications, DSML being by essence modeling languages domain experts are acquainted with. Our main objective is thus the development of an approach and implementing tool with which model engineers and domain experts can work hand in hand to specify and validate DSMLs. Achieving this goal requires the completion of the following main objectives:

O.1 Definition of an approach enabling the specification of DSMLs in Alloy.

The definition of a DSML validation process relying on Alloy analysis requires that the structure of Alloy DSML specifications is well defined.

O.2 Development of a new design process relying on domain experts' knowledge to validate Alloy specifications.

The main focus of this thesis is to enable the involvement of domain expert in the validation of DSML specifications. This new design process should thus rely on the intuitive concrete syntax of the DSML to bridge the gap between Alloy and domain experts.

O.3 Implementation of a tool based on the results obtained by completing O.1 and O.2.

This tool is meant to enable the validation of our approach.

The definition of DSMLs has been thoroughly studied in the past. Notably, Kleppe, in her book [41], provides an extensive state of the art on DSMLs. Specifically, she shows that a DSML can be expressed exclusively in terms of models and model transformations.

Alloy can be used to model systems and model transformations and has proven particularly effective at seamlessly validating designs, as highlighted in [42, 43]. Its ability to generate instances (example models *conforming* to the metamodel defined by the analyzed Alloy module) is a mechanism perfectly suited to involve domain experts in the validation process, provided that those instances are presented in an intuitive syntax. This is why this thesis focus on the fulfillment of the previously listed objectives relying mainly on Alloy. A drawback of using Alloy to specify every aspects of a DSML is when it comes to execute model transformations. Though Alloy can be used to specify model transformations, model transformations expressed in Alloy cannot be efficiently computed (as we will see in Chapter 3). Improving the usability of the tool developed in fulfillment of objective **O.3** hence necessitates the development of an Alloy-based solution (to fulfill objectives **O.1** and **O.2**) to the specification of efficiently computable model transformations. From this follows the last objective we propose to address in this thesis:

O.4 Development of an Alloy-based solution to the specification of efficiently computable model transformations.

1.3 Contributions

The research that has been conducted on the combined use of DSMLs and Alloy has led to several contributions. Those contributions are:

- in the domain of model transformations:

C.1 The definition of *F-Alloy*, a new model transformation language allowing the specification of efficiently computable model transformations. F-Alloy has the specificity of being a variant of Alloy in the sense that its syntax is a subset of the Alloy syntax and in the sense that its semantics only differ by the fact that, in F-Alloy, constraints related to the normal behavior of model transformations are left implicit. A translation procedure (from F-Alloy to Alloy) is defined as a sequence of constraints to be added to the F-Alloy specification. This simple translation defines the semantics of F-Alloy in terms of Alloy, making of F-Alloy a formal language. F-Alloy has been designed to enable the specification of *endogenous in-place*, *exogenous* and *compound* model transformations.

- C.2** The definition of hybrid analysis as a novel approach to the analysis of F-Alloy model transformation. Hybrid analysis relies on the Alloy analysis of the transformation’s source metamodel and on the F-Alloy interpretation of the transformation specification. It has been proven that hybrid analysis is equivalent to Alloy analysis in the sense that it yields the same set of instances, yet, complexity-wise, it has been shown that the completion time of a transformation’s hybrid analysis is reduced to the completion time of its source metamodel’s Alloy analysis.
- C.3** The design of a novel approach to the validation of model transformation called Visualization Based Validation (*VBV*) enabling the involvement of domain experts. It relies on Hybrid analysis to efficiently generate traces of F-Alloy model transformation specifications, and on a graphical concrete syntax specification (defined as a compound model transformation from the model transformation specification to validate to a Visual Language Model) to intuitively represent the trace’s source and target model. We show that this has as effect to enable the involvement of those domain experts familiar with the domain modeled by the transformation’s source and target metamodel.
- in the domain of Software Language Engineering:
- C.4** The definition of an approach to the engineering of DSMLs in Alloy. We show that DSMLs abstract syntax, concrete syntax and operational semantics can all be specified using Alloy (and F-Alloy for efficiency’s sake) in a way that allows, provided an abstract syntax’s instance, the seamless application of both concrete syntax and operational semantics.
- C.5** The design of an agile DSML engineering design process enabling at each design iteration the seamless validation, using Alloy analysis, of DSML components. To enable domain experts to get involved in this design process, validation relies on the concrete syntax and operational semantics defined to provide the domain experts with intuitive visualization and simulation features.
- C.6** The implementation of the *Lightning* tool, an eclipse plug-in allowing the specification and validation of DSMLs following our proposed approach. This tool is also packed with the F-Alloy interpreter and provides all the feature essential to the specification and usage of DSMLs, hence allowing its qualification as a full fledged language workbench.
- C.7** The application of *Lightning* to a real world case study. *Lightning* has been used to specify a robotic DSML called RPSL, following our DSML engineering design process with the creator of RPSL itself acting as domain expert. *Lightning* not only allowed to validate specifications of the RPSL language, but also was used to develop a framework to validate so called design alternatives (modeled by RPSL specifications.)

Relations between contributions and the research objectives they fulfill are drawn in Table 1.1. Table 1.1 also associates to each contribution the chapter in which it is presented in this thesis.

Research Objectives	Contributions	Chapters
O.1	C.4	Chapter 5
O.2	C.3 C.5	Chapter 4 Chapter 5
O.3	C.6 C.7	Chapter 6
O.4	C.1 C.2	Chapter 3 Chapter 4

Table 1.1: Table mapping each research objectives to the contribution fulfilling them and to the chapter in which those contributions are presented

1.4 Publications

The work described in this thesis has led to several journal and conference publications. In the following, we list and provide concise overviews of those publications in chronological order. For each listed publication, we refer the reader to the chapter of this thesis covering the same material.

2014. Domain-Specific Visualization of Alloy Instances, by *Loïc Gammaitoni and Pierre Kelsen* is published in the proceedings of the International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014) [44].

This paper demonstrates that big Alloy instances are generally hard to comprehend due to the genericness of the visualization provided by the Alloy analyzer. An approach to depict Alloy instances in a more intuitive manner is proposed. This approach is given in Section 5.2.2.

2014. Verifying Modelling Languages Using Lightning: a Case Study, by *Loïc Gammaitoni, Pierre Kelsen and Fabien Mathey* is published in the proceedings of the Workshop on Model-Driven Engineering, Verification and Validation (MoD-EVVA 2014) [45].

This paper gives the first hints that DSMLs can be defined in Alloy. It demonstrates how the Lightning tool can be used to define and verify the Structured Business Process Language. Material presented in this paper is covered by Chapter 5

2015. F-Alloy: An Alloy Based Model Transformation Language, by *Loïc Gammaitoni and Pierre Kelsen* published in the proceedings of the International Conference on Model Transformations (ICMT 2015) [46].

This paper introduces the F-Alloy model transformation language which is based on a subset of the Alloy language. It is given a translational semantics to Alloy, enabling the use of Alloy analysis for validation/verification' sake, as well as an interpretation procedure enabling efficient computations. Note that the notion of functional Alloy module is coined in this paper. This paper hence covers parts of Chapter 3.

2015. Designing Languages Using Lightning by *Loïc Gammaitoni, Pierre Kelsen and Christian Glodt* published in the proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015) [47].

This tool paper provides a more in-depth introduction of Lightning, an Alloy-based language workbench. It is mainly shown that Lightning supports the definition of

abstract syntax, concrete syntax and operational semantics of DSMLs using Alloy and F-Alloy. The Lightning tool is presented in Chapter 6.

2016. Agile Validation of Higher Order Transformations Using F-Alloy, by *Loïc Gammaitoni, Pierre Kelsen and Qin Ma* published in the proceedings of the International Conference on Theoretical Aspects of Software Engineering (TASE 2016) [48].

This paper coins the notion of hybrid analysis, a combination of Alloy analysis and F-Alloy interpretation enabling the efficient analysis of F-Alloy specifications. This is covered in Chapter 4.

2016. RPSL meets lightning: A model-based approach to design space exploration of robot perception systems, by *Loïc Gammaitoni and Nico Hochgeschwender* published in the proceedings of the Workshop on Simulation, Modeling, and Programming for Autonomous Robots Proceedings (SIMPACT 2016) [49].

This paper, an application of our work on a real world case study, proposes a design space exploration framework, based on Lightning, enabling robotic experts to systematically explore the design space of robot perception systems. This contribution is summarized in Section 6.2.

2017. Agile Validation of Model Transformations using Compound F-Alloy Specifications, by *Loïc Gammaitoni, Pierre Kelsen and Qin Ma* published in a special issue of the Science of Computer Programming (SCP) Journal [50].

In this extended version, proofs are more detailed and VBV, a new approach to the validation of model transformation based on compound model transformation, is introduced. This publication covers the same material as Chapter 4.

2017. F-Alloy: A Relational Model Transformation Language Based on Alloy, by *Loïc Gammaitoni, Pierre Kelsen* to be published in a special issue of the Software and Systems Modeling Journal.

This work presents an extension of F-Alloy enabling the expression of endogenous model transformations. This is covered in Chapter 3.

1.5 Thesis Outline

We now present the structure of this thesis.

Chapter 2 introduces the context of our work. It provides an overview of the research domains our work is ranging over and introduces the Alloy language as our work heavily relies on it.

Chapter 3 is dedicated to the introduction of a new language, F-Alloy that is based on Alloy and allow the specification of efficiently computable model transformations.

Chapter 4 presents an F-Alloy language extension allowing the specification and execution of so called compound transformations. This extension paves the way to a domain specific validation approach of F-Alloy transformations, named VBV, which is also presented in this chapter.

Chapter 5 presents an approach to the definition of DSMLs based exclusively on Alloy and F-alloy specifications and proposes a design process allowing to seamlessly find and fix design errors at the earliest stage of the language definition.

Chapter 6 presents the Lightning tool, implementing the previously introduced approach. A real world application of the tool is also given.

Chapter 7 concludes this thesis by a providing a summary of contributions and future work proposals.

Chapter 2

Background

The research synthesized in this work is situated at the junction of Model Driven Engineering and Software Language Engineering. We present those fields in Section 2.1 and 2.2, respectively. Our contributions relying heavily on Alloy, we provide an extensive introduction to this language in Section 2.3 and show how the language can be used to specify model transformations in Section 2.4.

2.1 Model Driven Engineering

During the last decade, mankind has become more and more dependent on software systems. Most of those systems are complex entities, that are difficult to be comprehended in their entirety [51]. It is thus natural that methodologies aiming at reducing the development, maintenance and verification complexity of such systems emerged. Model Driven Engineering (MDE) consists of those methodologies relying on representative abstractions called models [52].

2.1.1 Models

In the context of software engineering, models can be encountered in various forms in a multitude of different contexts. Their main use is to adapt the representation of a complex system putting forward essential information one might be interested in [51]. For this purpose, models are making abstraction of all details which are irrelevant for the particular purpose we want to use them for.

As we humans have limited perceptions and processing capabilities [53], we resort to models to describe and interact with our world even without us noticing. Sentences spoken or written in a given language are as many models of thoughts. Mathematical equations are used in physics, finance, and a plethora of other disciplines as models on which calculations and reasoning can be performed.

While the use of models in the software engineering field is not recent, a lot of interest was shown for this field in the last decade as modeling can play a key role in the creation of reliable software systems [54].

An interesting fact is that if a model can be used to represent a set of possible systems, it can also be used to represent a set of possible models. A model defining an abstract representation of models is called a metamodel. The Object Management Group [55] identified 3 levels of abstractions on top of a reality layer as depicted in Fig.2.1. At the top of this pyramid is the Meta Object Facility [56] (MOF), an OMG standard defining a set of essential concepts needed to define models (and sufficient to define itself, i.e., MOF is a metamodel of itself). MOF (M3) can hence be used to define modeling languages

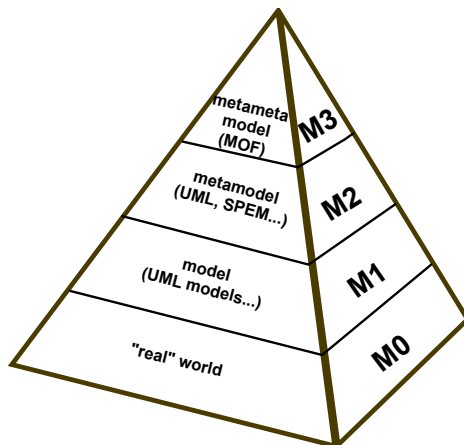


Figure 2.1: The OMG four-layer pyramid (taken from [1])

(M2) which can themselves be used to define models (M1) which are representations of systems in the real world (M0).

The Alloy language that will extensively be used in this paper has the particularity of spanning three levels. The real world (M0) is represented by Alloy instances. The model (M1) providing an abstract definition of those instances is given as an Alloy module. The set of possible Alloy modules (in M1) can be defined by an Alloy module (in M2).

2.1.2 Model Transformations

We have previously seen that models can be used to provide a concise representation of complex systems. Model transformations further pushes the usefulness of models by allowing the specification of operations to be performed on given models hence allowing the production of derived artifacts.

The structure of model transformations in their most common form is depicted in fig. 2.2.

A model transformation is written in a well suited *transformation language* conforming to, just like any modeling language, a meta-metamodel. The transformation itself is specified at the M2 level between the source and target meta-model. It generically describes how the transformation is to be executed given *source models* (conforming to the source metamodel) in order to obtain the expected *target models* (conforming to the target metamodel).

A variety of model transformation kinds have been developed to match the needs one might have when working with models [57]. In the following, we introduce a selected few types of model transformations that the reader might encounter in this work.

- **M2T**(Model to Text): A model transformation specification is said to be an M2T model transformation when its execution produces text rather than models. Those transformations are mainly used to generate code from given models. To explicitly declare that a model transformation is not M2T, one can qualify it as M2M (Model to Model) transformation. In this work, we focus on M2M model transformations.
- **Endogenous & Exogenous**: A model transformation is said to be *endogenous* when its source and target models conform to the same meta-model. It is said to be *exogenous* otherwise. Endogenous transformations are mainly used for refinement

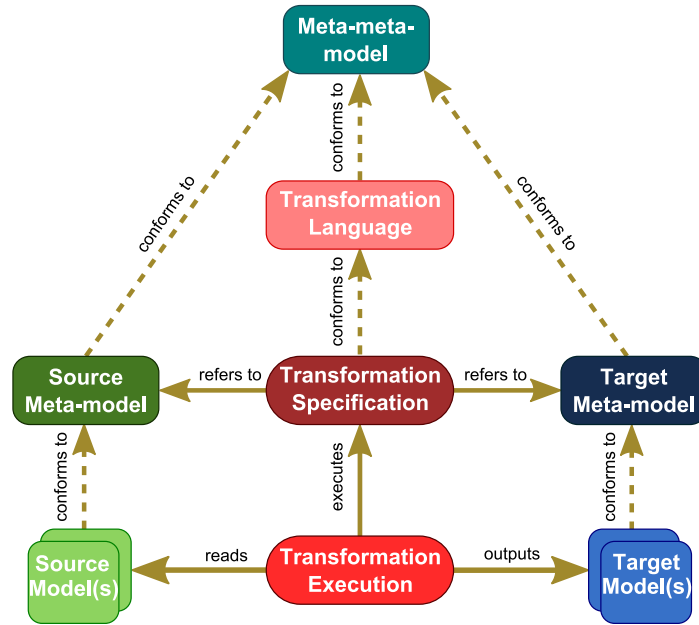


Figure 2.2: The big picture of model transformations (adaptation of [2] given in [3])

and refactoring purposes while exogenous transformation are used to export models into another formalism.

- **In-place & out-place:** A model transformation is said to be *in-place* if the target model is obtained by performing changes to the source model. A model transformation is said to be *out-place* if, on the contrary, it creates the target model from scratch, given the source model. In-place transformations commonly follow a *CRUD* approach [58] to the specification of changes to be brought to the source model. Hence, in-place transformation specifications are often expressed in terms of create, update and delete operations.
- **HOT (Higher Order Transformation):** Coined in [59], these are transformations whose source or target metamodels are transformation languages themselves.
- **Compound Model Transformation:** A model transformation whose source or target metamodels are transformation specifications themselves. To our knowledge no prior work explores their uses and properties. We discuss this kind of transformations in chapter 4 when introducing Visualization Based Validation as a novel approach to validate model transformations.
- **Bx (Bidirectional):** Bidirectional model transformations [60] have the property of being executable both from source to target and from target to source models with consistent results.

In this work, we restrict ourselves to the study of exogenous out-place and endogenous in-place model transformations. We ignore endogenous out-place transformations as we regard them as a special application of exogenous out-place transformations, where the target meta-model is a duplicate of the source metamodel, considered as different though declaring the exact same concepts. Throughout this thesis, we therefore mean by endogenous transformation, if not stated otherwise, endogenous in-place transformation.

2.2 Software Language Engineering

The Software Language Engineering (SLE) research field aims at developing techniques to ease the creation, verification and usage of domain specific languages [61].

2.2.1 Domain Specific Languages

A *Domain Specific Language* (DSL) is a language meant to be used on a restricted set of problems [62]. The specificity of those languages enable their creators – namely, *language engineers* – to come up with very lightweight and intuitive syntaxes that users of the language – namely, *domain experts* – can use with no or little prior training. One of the challenges faced by language engineers when developing a new DSL is thus to get familiar with the domain knowledge so as to leave it implicit to the user [63].

Interaction between language engineers and domain expert is thus manifest in the process of defining a new DSL.

We note that this thesis focuses on the engineering of those DSLs aimed at representing certain aspects of a system by models. This class of DSLs is called *Domain-specific modeling languages (DSMLs)*¹ (Domain Specific Modeling Languages).

A DSL definition generally consists of [61]:

- an *abstract syntax* defining the set of models that can be expressed in that language.
- a *concrete syntax* defining the notation with which models of the language are to be expressed.
- a *semantics*, giving a meaning to models expressed in that language.

We introduce those components in the following three subsections.

2.2.2 Abstract Syntax

The abstract syntax of a language defines the structure, in terms of concepts and relations, of models expressible by the language. Abstract syntax is commonly induced for textual DSLs by the accompanied Abstract Syntax Tree (AST) used to parse textual inputs, i.e., identify the concept embodied by each sequence of characters. Yet one can also use metamodels to define the static structure of an ASM. This latter approach is used in the definition of DSMLs where the need of word by word parsing, inherent to textual DSLs, is inexistent. The abstract syntax then defines the set of all valid models expressible by the language. We call those models *language models*.

2.2.3 Concrete Syntax

The concrete syntax defines how language models are to be visualized and how one is meant to edit a language model through its visualization.

The concrete syntax of a textual DSL is often defined at the same time as the abstract syntax using Backus-Naur Form (BNF) notation from which existing tools (e.g. xText [64]) allows the generation of both AST and editor.

For DSMLs, a definition of the graphical symbols to be used in the visualization of language models is needed. It can generally be provided as a meta-model that we propose to call *Visual Language Meta-model (VLM)*. The concrete syntax can then be defined as a transformation from abstract syntax to VLM. Such a transformation is

¹Diagrammatic DSML is a term that can also be encountered

generally bidirectional to support edition of language models through their graphical representation.

2.2.4 Semantics

There exist several approaches to providing a meaning to a DSL. The semantics of a DSL is said to be

- **Operational** if it has been given the definition of an abstract state machine showing how language models are meant to be executed
- **Denotational** if each syntactic construct of the language has its meaning defined in terms of mathematical objects
- **Translational** if has been provided a transformation from its abstract syntax to the abstract syntax of a language whose semantics is well defined.
- **Pragmatic** if the language comes with a tool whose behavior dictates the meaning of each language model.

2.2.5 Language Workbenches

We give here a basic introduction of the term *language workbench*. More details are provided in Section 6.3, where our own tool Lightning is compared to existing language workbenches.

The term “language workbench” was made popular by Martin Fowler [65]; it denotes a tool that supports the efficient definition, reuse and composition of languages and their IDEs [5].

Those tools have generally in common the following set of features:

1. editor support for the notation used to define concepts of the language and their representation, i.e., the abstract and concrete syntax, respectively.
2. generation of editor from defined language, allowing the assisted creation and edition of language models.
3. means to express the semantics of the language, through the generation of code, translation to another language whose semantics is well-defined, or direct interpretation/simulation of language models.

2.3 The Alloy Language

Alloy [24] is a formal language based on a first-order relational logic with transitive closure. It is based on a small set of core concepts, the main one being mathematical relations. It was originally developed to support agile modeling of software designs. It does this by allowing fully automatic analysis of software design models using SAT solving. To reduce the computational complexity, Alloy performs SAT solving in a finite domain bounded by cardinalities associated to each concept declared. Those cardinalities are called *scopes*.

In the context of model driven software engineering, Alloy and its accompanying tool, the Alloy Analyzer, have been used to validate properties of models [33–36] and model transformations [42, 66]. In this section we provide an extensive introduction to this language.

2.3.1 An Informal Alloy Introduction Based on Ecore

Before introducing in detail the Alloy notions necessary to the good understanding of our work, we draw, for the sake of our readers familiar with Ecore, a parallel between Alloy and Ecore's main concepts.

- Alloy modules correspond to Ecore metamodels enhanced with OCL in the sense that they are entities used to define structural constructs and their well-formedness rules
- Signatures correspond to EClass in the sense that they allow the definition of concepts. Signatures can also be abstract and inherited. Signatures, to the contrary of EClasses, do not support multiple inheritance per-se but support the broader notion of set inclusion: each signature representing a set of atoms in Alloy, a signature can be defined as a subset of several others. Note that when a signature *A* extends a signature *B* in Alloy it does not only enforce that atoms in *A* are contained in *B*. It also enforces that the set of atoms defined by *A* is disjoint from any other set of atoms defined by signatures extending the same class as *A* (in short, inheritance declared using the `extend` keyword is a stronger notion than set inclusion defined using the `in` keyword).
- Fields are more expressive than any corresponding EStructuralFeature (EReference, EAttribute, ...) in the sense that they allow the definition of relations of any arity. Yet a field has no properties as opposed to, *e.g.*, EReferences, hence properties like containment or EOpposite are to be defined through constraints.
- Alloy Instances are the equivalent of Ecore instantiations represented by an XMI file.
- Atoms and Tuples are elements of an Alloy instance just like objects and links are of an Ecore instance.

2.3.2 Alloy Modules and Instances

A metamodel can be designed in Alloy in one or several *Alloy modules*, each *module* being associated to a single (.als) file.

An Alloy module optionally starts by a module declaration enabling elements herein declared to be reused by other modules.

The module declaration starts by the keyword `module` followed by a relative path to the file the module is contained in. This path defines the scope in which the module can be imported – i.e., it is available for import for any other Alloy module contained in the root of that path. An import is then performed by using the `open` keyword followed by the relative path of the module to import.

As an example, let us consider the directory tree given in fig. 2.3:

- In order to allow the Alloy module `a1` to import module `a3` but not `a2`, then the module declaration of `a1` and `a3` should be `module A/A1/a1` and `module A/A2/a3`, respectively. The module `a1` would then contain the import instruction `open A/A2/a3`.
- In order to allow `a2` to import both `a1` and (transitively) `a3`, it is necessary that the three module declarations contain the common parent directory `Project`. The header of those three modules can thus be as depicted in Fig. 2.4.

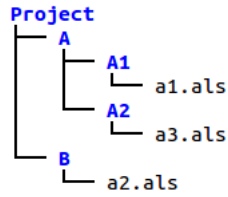


Figure 2.3: Example of a directory tree containing Alloy files

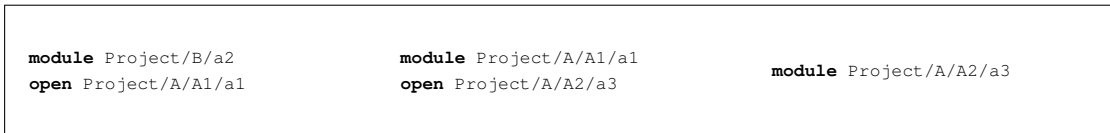


Figure 2.4: Headers of Alloy module a1, a2 and a3 corresponding to the directory tree given in fig.2.3

Importing a module enables the use of all constructs declared in it: *signatures*, *fields*, *facts*, *predicates*, *functions*, *assertions*, *let* expressions and *commands*. All those terms will be introduced in the following.

Given an Alloy module, the Alloy Analyzer is a tool yielding a set of *Alloy instances* – models whose elements are typed by concepts and relations of the metamodel defined by the Alloy module and that satisfy constraints of the module.

An instance is called *counter-example*, if it is obtained by checking an assertion. Counter-examples are instances in which the checked assertion is violated.

Note that instances obtained by analysis of an Alloy module *a* are generally called “instances of *a*” and can be shortened “*a*-instances”.

In the following we give detailed information on declarable constructs composing an Alloy module while giving hints on how they impact the analysis carried by the Alloy Analyzer.

2.3.3 Signatures and Fields

An Alloy module is primarily composed of signatures and fields, which single-handedly define the set of elements that can compose any Alloy instance obtainable by the Alloy analyzer: instances are composed of non-dividable entities called *atoms*, whose type is given by signatures, and of atom *tuples*, whose type is given by fields.

The declaration of a signature is made using the keyword **sig** followed by the name of the signature and a block containing a set of field declarations relating this signature with others. A signature can also *extend* another signature to define subtypes. A set of modifiers can precede any signature declaration:

- **Multiplicity keywords** such as `none`, `one` and `some` enforce the number of atoms typed by the declared signature to be at most one, exactly one or at least one, respectively. Without multiplicity keyword, there can be any number of atoms typed by the signatures.
- **The `abstract` keyword** enforces that no atom is directly typed by that signature¹.

¹This modifier is ignored by the analyzer if the signature on which it is applied is not extended by

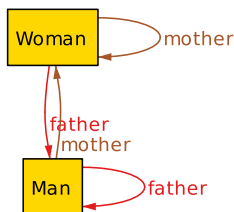


Figure 2.5: Alloy instance of the Alloy module given in Listing 2.1

The declaration of a field consists of a label used as identifier, followed by a sequence of arrow-separated signatures. Multiplicity can be specified at each arrow end.

In Listing 2.1, we provide a sample Alloy module describing concepts and relations of a family.

```

abstract sig Person{
  mother: lone Woman,
  father: lone Man
}
sig Man extends Person{}
sig Woman extends Person{}

```

Listing 2.1: sample of an Alloy module defining parental relations between persons

Instances of this module will be composed of atoms typed by either the Man or the Woman signatures and of tuples typed by either the mother or the father fields. An example of such instance is given in fig.2.5. As suggested by this instance, structural information is not sufficient to accurately define families. Constraints are needed to enforce a set of properties on instances, e.g., a woman cannot be her own mother nor the mother of her father.

2.3.4 Facts

Facts are used to specify constraints, properties that should always hold in instances of the Alloy module they are declared in.

A fact declaration starts by the keyword **fact** followed optionally by a name (used for documentation purpose solely) and a block containing a boolean valued *Alloy expression*.

As an example, the fact depicted in Listing 2.2 would prevent instances of the Alloy module given in Listing 2.1 to contain persons who are their own ancestors.

```

fact noSelfAncestor{
  all p:Person | p not in p.^(father+mother)
}

```

Listing 2.2: An Alloy fact stating that no person should be its own ancestor

When specifying invariants proper to a given signature, it is possible to declare what is called a *signature fact*. A signature fact takes the form of a block succeeding a chosen signature which automatically becomes its context. By context, we mean that the invariant defined in the fact is applying to each atom typed by the given signature. We provide in Listing 2.3 a refined version of the `noSelfAncestor` fact, this time expressed as a signature fact of `Person`. We note that in a signature fact, the key word `this` is used to refer to the context (any given person) and the operator `@` is used to refer to fields declared in the signature `Person` outside of the context.

any other signature.

```

abstract sig Person{
  mother: lone Woman,
  father: lone Man
}{
  this not in this.^(@father+@mother)
}

```

Listing 2.3: An example of signature fact expressing the same invariant as the `noSelfAncestor` fact given in Listing 2.2

2.3.5 Predicates and Functions

Predicates and functions are both parameterizable Alloy expressions, the former being boolean-valued and the latter being set-valued (return a set of atoms or tuples of atoms).

Predicates and functions are declared using the keyword **pred** and **fun**, respectively, followed by an identifier, optional parameters and a block containing Alloy expressions (with possible occurrences of those previously mentioned parameters).

Those parametrized expressions are illustrated in Listing 2.4 where the function `getSiblings` returns the set of persons who have the same father or mother than the person given in parameter and the predicate `atLeastOneGrandPa` holds in any instances containing a grandfather.

```

fun getSiblings(p:Person): set Person{
  p.(mother+father).~(mother+father) - p
}

pred atLeastOneGrandPa{
  some m:Man | m.~father.~(mother+father) != none
}

```

Listing 2.4: example of Alloy function and predicate

2.3.6 Assertions and Commands

Assertions are special predicates (declared with the keyword **assert**) in the sense that they are used solely in commands. An example of an assertion is given in Listing 2.5.

```

assert GrandPaTest{
  atLeastOneGrandPa implies #Person <= 3
}

```

Listing 2.5: example of an Alloy assertion asserting that instances containing a grandpa should have at least 3 Persons

A command is mandatory for the Alloy analyzer to be able to generate instances for a given Alloy module. It starts by a keyword providing information on the kind of analysis to be performed: Those are :

- **run**: given a predicate, generate a sample of instances in which the predicate holds.
- **check**: given an assertion, generate a sample of counter-examples in which the assertion is violated.

Both kind of commands should also provide information on the domain space in which the analysis is to be performed, the *Alloy analysis* being decidable only because it is performed on a finite domain.

This is achieved by associating a scope to each signature of the module, i.e., an upper bound to the number of atoms typed by each signature of the module.

By default the scope for all signatures is set to 3. This global scope can be modified using the keyword **for**. It is also possible to assign a scope for each signature separately or provide both global and specific scopes using the **but** keyword. The use of commands is illustrated in Listing 2.6.

```

/* looking for instances with 0 to 2 Man and 0 to 5 Person */
run{} for 5 but 2 Man

/* looking for instances with 0 to 4 Man and exactly 2 Woman */
run{} for 4 Man, exactly 2 Woman

/* looking for instances with 0 to 5 Person in which the predicate atLeastOneGrandPa holds */
run atLeastOneGrandPa for 5

/* looking for counter examples with 0 to 10 Person in which the assertion GrandPaTest is violated */
check GrandPaTest for 10

```

Listing 2.6: example of Alloy commands (commented)

2.4 Expressing Model Transformations in Alloy

Model transformation playing a central role in model driven engineering approaches, we present in this section a generic approach to the specification of model transformations in Alloy. We then discuss it under the light of related work before synthesizing a list of benefits and limitations that the use of Alloy brings when used in the specification of model transformations.

We start out by adapting the model transformation terminology given in Section 2.1.2 to the Alloy world, namely, Alloy modules being specification of metamodels and instances being representation of models, we call:

- *Source module and target module* the source and target metamodels of a model transformation specified in Alloy
- *Source and target instances* the source and target model of a model transformation specified as Alloy modules

2.4.1 A Generic Approach to the Expression of Model Transformations in Alloy

Overview

In Section 2.3, we have seen that relations between concepts are defined in Alloy through the declaration of fields.

A natural approach to define a transformation from a source module a_{src} to a target module a_{dst} as an Alloy module a is thus to declare, in a signature, fields relating signatures of a_{src} to signatures of a_{dst} . Those fields, that we call *mappings* from now on, should then be suitably constrained so that in any a -instance, the presence of certain a_{src} -elements (atoms and tuples typed by signatures or fields declared in a_{src}) enforces the presence of their expected images (w.r.t. the transformation defined). Given such an Alloy module a , executing the transformation it defines on an a_{src} -instance x_{src} consists in using the Alloy analyzer to find a -instances in which the a_{src} -sub-instance is x_{src} . This approach is illustrated in Fig.2.6.

In this figure, we see that any instance conforming to a (white rounded rectangle) is composed of two sub-instances representing the source instance (green rounded rectangle) and its corresponding target instance (red rounded rectangle). Tuples typed after the

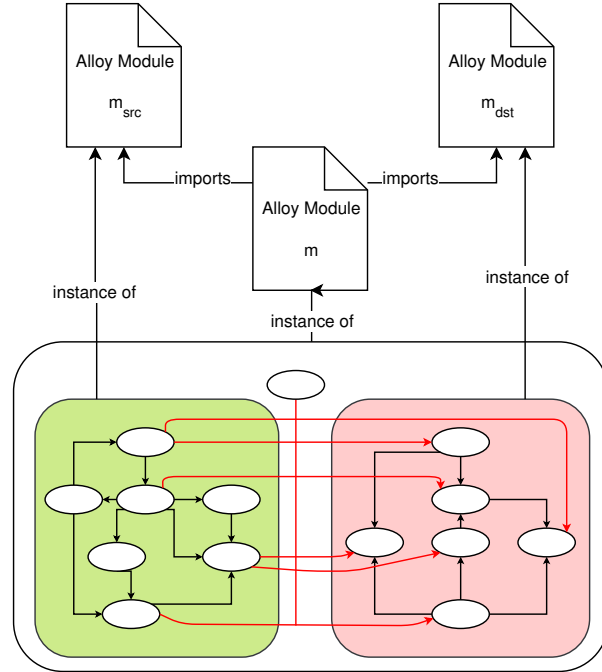


Figure 2.6: Illustration of the approach to represent a model transformation in Alloy

mappings declared in a (red arrows) relate the atom in the source instance to their image in the target instance. The constraints to be added to an Alloy module so that its instances denote effectively an execution of the transformation defined can be categorized as follows: Given an Alloy module defining a transformation from a_{src} to a_{dst} :

C_PRE : Constraints defining when atoms of the source instance are to be mapped to atoms of the target instance. They can be seen as defining a “preconditions” for mappings.

C_POST : Constraints defining how atoms of the target instance relate to each other. They can be seen as defining a “postcondition” for mappings.

C_TRACE : Constraints ensuring that each instance represents an actual execution of the transformation defined, that is, one can, from it, obtain the source instance, the target instance, and *traceability links*, showing from which mapping and for which elements of the source instance an element of the target instance has been created.

We note that this general approach can be applied to both exogenous and endogenous transformations. In the next two subsections, we give a more specific description of how the **C_TRACE** constraints can be implemented with respect to the nature of the transformation defined.

Expressing Exogenous Transformations in Alloy

Let us consider an Alloy module a defining an exogenous transformation from a_{src} to a_{dst} following the previously given approach. To ensure that any instance of a effectively represents an execution of the transformation defined, it is necessary to prevent the presence of a_{dst} elements which have no counterpart in a_{src} (as for a transformation

to be relevant, the yielded target instance should depend exclusively on the source instance given). The **C_TRACE** constraints enforcing a -instances to effectively represent transformation executions should thus be for exogenous transformation:

C_TRACE_EX Constraints ensuring that atoms typed by signatures declared in a_{dst} are all part of a mapping (hence preventing the presence of a_{dst} atoms without a_{src} counterparts).

We then have, given an a -instance x :

- the source instance of the transformation being the set of elements in x typed by a_{src} signatures and fields
- the target instance of the transformation being the set of elements in x typed by the a_{dst} signatures and fields
- the traceability links being the tuples in x typed by mappings declared in a .

Expressing Endogenous Transformations in Alloy

The specification of endogenous transformations differ from the specification of exogenous transformations in the sense that mappings do not specify a correspondence between elements in the source and target instances but specify operations to be applied on the source instance in order to obtain the target instance. Those operations, identified in Section 2.1 as create, update and delete operations, can also be represented by mappings, with the following effects:

- a mapping representing a create operation maps elements of the source instance satisfying a certain precondition to a new element respecting a certain postcondition
- a mapping representing an update operation maps elements to be updated (identified by a precondition) to a new element respecting a certain postcondition, with the effect that the latter replace the former.
- a mapping representing a delete operation marks those element identified by a given precondition for deletion (deletion is the implicit postcondition).

We choose, in order to differentiate mappings by their purpose, that is, by the nature of the operation they represent, to declare them in three different signatures, each of them containing mappings of a same nature. For that reason, our Alloy-based representations of endogenous model transformations will always contain three signatures, namely `CREATE`, `UPDATE` and `DELETE`, each containing mapping declaration of their respective kind.

In this context, the **C_TRACE** constraints enforce instances of the Alloy module defining an endogenous transformation to effectively represent transformation executions as a source instance and a set of “operation traces” from which it is possible to derive the target instance. These constraints can be detailed as follows. Considering an endogenous transformation refining a_{src} -instances defined in an Alloy module a opening a_{src} :

C_TRACE_EN_1 : To enable a clear separation between source and target instance, elements of the source instance cannot be “created from” or “be an updated version of” other source elements.

C_TRACE_EN_2 : To enforce source and target instance derivable from a -instances to be a_{src} -instances, constraints enforcing a_{src} -invariants to hold in the source and target instance are needed.

We then have, given an a -instance x :

- the source instance of the transformation being the set of elements in x minus those tuples typed by mappings minus atoms which are “created from” or “an updated version of” source elements as specified by CREATE and UPDATE mappings.
- the target instance of the transformation being the source instance, minus atoms marked for deletion by DELETE mappings, in which atoms marked for update by UPDATE mappings are replaced by their “updated version” and in which atoms “created from” source elements by CREATE mappings are present.
- the traceability links being the tuples in x typed by mappings declared in a .

2.4.2 Related Work on the Use of Alloy in the Specification of Model Transformations

We introduced in the previous section a generic approach towards the specification of model transformations in Alloy. Similar approaches, exploiting the relational nature of Alloy, have been used in several related works to specify exogenous model transformations (to our knowledge, no prior work studied the use of Alloy in the specification of endogenous in-place model transformations).

We discuss those related works in the following.

In [42], Anastasakis et al. use Alloy to analyze the correctness of model transformations. They resort to their tool UML2Alloy [67] to transform the source and target metamodels into Alloy and manually translate QVT transformation rules into an Alloy module containing mappings and predicates. It becomes then possible to use the Alloy analyzer to verify transformation properties by either generating random traces to be validated or by checking assertions. While this work opens the possibility of applying Alloy analysis to model transformations, the manual specification of a QVT model transformation in Alloy is an error prone exercise. In a similar line of work Baresi et al. [66] use Alloy to represent graph transformations represented in the AGG formalism. Though authors give hints on how AGG specifications are to be translated into an Alloy module, the translation is not formally defined and no translators have been implemented. To our knowledge, the first work supporting the fully automatic verification, based on Alloy, of model transformations expressed in another model transformation language is [43], where Alloy is used as an execution engine for QVTr model transformations. Indeed, this work also describe an approach to translate QVTr specification to Alloy and a tool named echo [68] implements this translation.

2.4.3 Benefits and Limitations of Alloy in the Specification of Model Transformations

All those presented related works motivate their use of Alloy by the following benefits:

- B.1** Alloy is a formal language [69] allowing the precise specification of systems
- B.2** The automatic analysis provided by the Alloy Analyzer is a lightweight yet effective verification mean [36, 70, 71].

However, approaches relying on Alloy analysis suffer limitations inherent to the use of the Alloy Analyzer. Those limitations are the following:

- L.1** Despite many advances in the performance of SAT solvers [72], the analysis of Alloy specifications does not scale. Indeed, while the analysis of simple models in a small scope terminates almost instantaneously, termination is not guaranteed when the model analyzed is complex¹ or requires a larger scope to find suitable instances (increasing scopes leading to a combinatorial explosion).
- L.2** The task of finding a minimal sufficient scope is by itself non-trivial. This is particularly problematic for complex Alloy specifications as every constraints and fields are to be taken into considerations while defining the scope of each signatures. As of today, no heuristics currently exist to automatize this task.
- L.3** Enforcing “transformation behaviors” require the addition of verbose constraints, we identify in this work as **C_PRE**, **C_POST** and **C_TRACE** constraints. Those constraints can prove hard to maintain in the case of any changes to the transformation specification.

Alloy modules defining a model transformation are inherently complex as they are a combination of two modules with extra constraints.

The previously listed limitations should thus be addressed before making Alloy a viable language alternative to specify model transformations.

In the next section, we propose a solution tackling the scalability issues of Alloy while retaining the aforementioned benefits. This solution takes the form of an Alloy-based DSL called *F-Alloy*.

¹by the amount of concept declared and the complexity of constraints expressed

Chapter 3

The F-Alloy Language

In this chapter we introduce a new language called F-Alloy, designed for the sole purpose of specifying model transformations. This language, based on Alloy, is meant to retain in the context of model transformation specification all the benefits of Alloy specifications while overcoming its limitations. This chapter is structured as follows: after motivating the design of the F-Alloy language in Section 3.1, we present in Section 3.2 two model transformation case studies along with their implementations in both F-Alloy and other existing model transformation languages. Through the comparison of those implementations, readers can get a first intuition of F-Alloy’s syntax and semantics. This intuition should facilitate the comprehension of extensive details on the syntax and (translational) semantics of the language provided in Sections 3.3 and 3.4, respectively. We then show in Section 3.5 how F-Alloy specifications can be efficiently computed via a process called interpretation, and compare in Section 3.6 the performance of F-Alloy’s interpretation with the performance of Alloy analysis and the execution performance of other transformation languages. Finally, we conclude this chapter by discussing some related work on model transformations in Section 3.7 and summarizing this chapter’s contributions in Section 3.8.

3.1 From Alloy to F-Alloy

In this Section, we motivate the creation of a new language called F-Alloy to enable the specification in an Alloy syntax of efficiently computable model transformations. Also, we give an overview of the desired expressiveness of this new language F-Alloy by defining the set of Alloy modules F-Alloy aims at specifying, the so called functional Alloy modules.

3.1.1 Motivation

We have seen in Section 2.3 that the formal language Alloy is designed to define and reason about relations. Model transformations embodying a set of relations – from elements of their source model to elements of their target model –, Alloy is a suitable language to specify model transformations (as illustrated in Section 2.4). In Section 2.4.3, we have identified a set of limitations and benefits related to the use of Alloy in the specification of model transformations, namely:

- L.1** Despite many advances in the performance of SAT solvers [72], the analysis of Alloy specifications does not scale. Indeed, while the analysis of simple models in a small scope terminates almost instantaneously, termination is not guaranteed when the model analyzed is complex¹ or requires a larger scope to find suitable instances

¹by the amount of concept declared and the complexity of constraints expressed

(increasing scopes leading to a combinatorial explosion).

L.2 The task of finding a minimal sufficient scope is by itself non-trivial. This is particularly problematic for complex Alloy specifications as every constraints and fields are to be taken into considerations while defining the scope of each signatures. As of today, no heuristics currently exist to automatize this task.

L.3 Enforcing “transformation behaviors” require the addition of verbose constraints, identified in this chapter as **C_PRE**, **C_POST** and **C_TRACE** constraints. Those constraints can prove hard to maintain in the case of any changes to the transformation specification.

B.1 Alloy is a formal language [69] allowing the precise specification of systems

B.2 The automatic analysis provided by the Alloy Analyzer is a lightweight yet effective verification mean [36, 70, 71].

In order to overcome those limitations while still retaining the benefits, we are interested in designing a domain specific language (to tackle limitation **L.3**) allowing the specification of efficiently computable model transformations (to tackle limitations **L.1** and **L.2**). This new language, named F-Alloy, is based on a subset of the syntax of Alloy but its associated semantics differs in the sense that every Alloy construct is interpreted in the context of model transformations. This semantics “alteration” can be translated into explicit Alloy constraints hence allowing to easily translate F-Alloy specifications into Alloy and thus retaining Alloy benefits **B.1** and **B.2**.

Knowing that the preservation of Alloy’s benefits is achieved through a well defined translation, one could wonder if designing a new language was really necessary, as there exists a plethora of efficiently computable model transformation formalisms (e.g., ATL [73], TGG [74], QVTr [75]). Two main motives drove us to develop F-Alloy instead of reusing existing model transformation languages:

1. While the definition of translations between formalisms is a non-trivial (and hence error-prone) exercise – due to variation in the expressiveness of each language [76, 77], and the difference in approaches followed by each formalism ([78, 79]) – the F-Alloy language was designed so as to allow the definition of a simpler translation into Alloy. This simplicity is achieved by reusing structural concepts of Alloy and by differing from Alloy’s semantics solely by leaving the constraints presented in Section 2.4.1 implicit. Those constraints can be generated from any F-Alloy specification in a fairly straightforward fashion (as we will see in Section 3.4.2).
2. In this thesis, we are interested in defining an approach to the engineering of DSML based on Alloy. It is thus assumed that any user of our approach is well acquainted with the Alloy language. The fact that F-Alloy reuses the syntax of Alloy while only slightly altering its meaning should enable any Alloy user to intuitively specify model transformations. On the other hand, no assumption can be made on the familiarity of potential users of our approach with standard model transformation languages.

For the computation of F-Alloy specification to be efficient, we want them to be interpreted rather than analyzed, so that computation is performed incrementally and in a straightforward manner.

To achieve this goal, we restrict the set of model transformations we aim at specifying in F-Alloy to non-deterministic single output model transformations, i.e., *functions*.

In this section, we first introduce a simple mathematical framework allowing to reason about Alloy modules and instances, before introducing the concept of *functional Alloy module*, i.e., an Alloy module defining functions. In the next Chapter, we provide a detailed introduction of F-Alloy, a DSL allowing the specification of functional Alloy modules.

3.1.2 A Mathematical Framework to Reason About Alloy Modules and Instances

We recall from Section 2.3 that a metamodel can be expressed in one or several Alloy modules, each module being associated to a single file. A module may *import* other modules, in which case the importing module can use features of the imported modules.

We formalize those notions of module, signature and fields, already presented informally in Section 2.3, as follows:

Definition 1 (Alloy Module, Signature, Field). *An Alloy module is a tuple (S, F, φ) with S and F being the sets of signatures and fields declared in the module or any of its (recursively) imported modules, respectively. Signatures may be defined as subsignatures of other signatures (using the `extends` keyword). Fields of F have as type a sequence of signatures in S , the first one being the signature that contains it. φ is a first-order logic formula (possibly containing the transitive closure operators¹ $\hat{\ }^1$ and \ast) representing the set of constraints, called facts, expressed in the module.*

Considering now A , a set of indivisible entities called *atoms*, T , a set of atom tuples, and a module $a = (S, F, \varphi)$, we call *typed atoms* pairs (x, s) where $x \in A$ and $s \in S$. A typed atom (x, s) is also denoted x^s (read “atom x of type s ”). A *typed tuple* is a pair (t, f) where $t \in T$ and $f \in F$. A typed tuple (t, f) is also denoted t^f (read “tuple t of type f ”). Note that for a typed tuple t^f the following needs to hold: if the type of the field is (X_1, \dots, X_n) , then the i -th component of the tuple needs to have as type X_i or a (direct or indirect) subsignature of X_i .

We call x^s an *s-atom* and t^f an *f-tuple* and extend the superscript notation so that sets of s-atoms A and of f-tuples T are denoted A^s and T^f , respectively.

Definition 2 (Alloy Instance). *An Alloy instance of an Alloy module a , also called a -instance, is a triplet $x = (X, Y, a)$ where $a = (S, F, \varphi)$, X is a set of atoms typed by signatures of a and Y is a set of tuples typed by fields of a and composed of atoms in X . We write $x \models \varphi$ if an instance x of a satisfies φ and call *valid instances*² of a the subset of instances of a satisfying φ .*

We denote the set of valid instances of a by $\mathcal{Z}(a)$. Formally:

$$\mathcal{Z}(a) = \{(X, Y, a) \mid \forall x^s \in X, s \in S \wedge \forall y^f \in Y, f \in F \wedge (X, Y, a) \models \varphi\}$$

An instance (X, Y, a) is an *(a)-sub-instance* of (X', Y', a') if $X \subseteq X'$, $Y \subseteq Y'$ and a' is equal to a or a' imports a .

Alloy comes with its dedicated tool, the Alloy analyzer, enabling the automatic analysis of Alloy modules. This analysis returns for a given Alloy module a the subset of valid instances of $\mathcal{Z}(a)$ that fit within a given scope s . To simplify the notation and the subsequent reasonings using it, we consider only global scopes, that is, s is a natural

¹ R returns the smallest relation R' containing R and being transitive while $\ast R$ returns the smallest relation R' containing R and being both transitive and reflexive

²While in Alloy any instance is valid (as obtained by Alloy analysis), we relax this assumption as we will reason about instances built by interpretation whose validity is not ensured

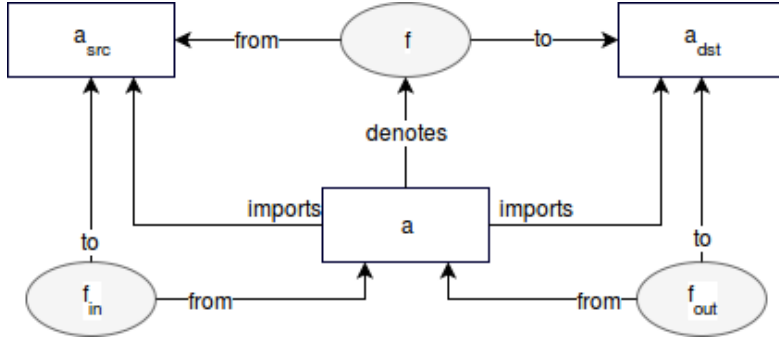


Figure 3.1: Illustration of the usage of transformation functions w.r.t. the definition of functional Alloy modules

number defining the maximal occurrence of atoms of each type¹. We then use the following notation to denote the set of valid a -instance obtained by Alloy analysis of a in a scope s .

Notation 1 (Alloy Analysis). *We denote the set of instances obtained by Alloy analysis of an Alloy module a within a global scope $s \in \mathbb{N}$ by $\mathcal{Z}(a, s)$.*

We then have $\mathcal{Z}(a) = \bigcup_{s=0}^{\infty} \mathcal{Z}(a, s)$.

3.1.3 Functional Alloy Modules

We now introduce a set of concepts and notation used to reason about model transformations expressed in Alloy.

We start by introducing the notion of *transformation functions* which may be viewed as the mathematical representation of a deterministic single-output model transformation expressed in terms of Alloy instances.

Definition 3 (Transformation Function). *Let a and a' be two Alloy modules. A transformation function f from a to a' , noted $f : \mathcal{Z}(a) \rightarrow \mathcal{Z}(a')$, is a function that takes as input a valid a -instance and returns as output a valid a' -instance.*

We then call *functional Alloy modules* Alloy modules specifying such transformation functions. We define them as follows:

Definition 4 (Functional Alloy Module). *Let a , a_{src} and a_{dst} be Alloy modules² and let $f_{in} : \mathcal{Z}(a) \rightarrow \mathcal{Z}(a_{src})$ and $f_{out} : \mathcal{Z}(a) \rightarrow \mathcal{Z}(a_{dst})$ be two transformation functions. We say that a is a functional Alloy module with respect to f_{in} and f_{out} if the following holds:*

$$\forall x, x' \in \mathcal{Z}(a), f_{in}(x) = f_{in}(x') \implies f_{out}(x) = f_{out}(x')$$

We then say that a specifies a (possibly partial) transformation function $f : \mathcal{Z}(a_{src}) \rightarrow \mathcal{Z}(a_{dst})$, such that

$$\forall x \in \mathcal{Z}(a), f(f_{in}(x)) = f_{out}(x)$$

We typically use Definition 4 in the following context: module a imports a_{src} and a_{dst} and define a transformation from the former to the latter. Function f_{in} and f_{out} then

¹We thus exclude for convenience (and as it does not have an impact on future reasonings), cases where scopes are defined individually for each signature

²with the possibility that $a_{src} = a_{dst}$

return given an a -instance the a_{src} and a_{dst} sub-instance corresponding to the source instance and target instance of the transformation expressed in a , respectively. We illustrate this use of those functions in Fig. 3.1.

We note that the nature of the transformation (endogenous or exogenous) influences the way those source and target instances are obtained from an a -instance. An informal description of procedures yielding the source and target instance from instances of an exogenous or endogenous model transformation specification has already been given in Section 2.4.1. A formal definition of the transformation functions denoting those procedures will be given in Section 3.4.3.

To give a first intuition of the behaviors and benefits of the F-Alloy language – designed to succinctly specify functional Alloy modules – we present in the next section two model transformations and their implementations in both F-Alloy and languages well known by the model transformation community.

3.2 Case Studies

We present in this section two model transformations used as case studies to provide a soft introduction to F-Alloy — by drawing a parallel between F-Alloy implementations and implementations in languages well-known to the model transformation community — and as a mean to evaluate and illustrate our model-transformation-related contributions.

The first model transformation — referred to as `CD2RDBMS` — is an exogenous model transformation from Class Diagram to Relational Database Management System. The second model transformation — referred to as `CDRefinement` — is an endogenous model transformation on Class Diagrams.

We start by introducing the source and target metamodels of those transformations using both Alloy and Ecore notations.

3.2.1 The CD and RDBMS Alloy Modules

The CD and RDBMS modules in Listing 3.1 and 3.2, define the same structure than the metamodels given in Figure 3.2 and 3.3. In those modules, extra well-formedness constraints were added such as:

- In the CD module,
 - `disj`, line 5, enforces that different `CDElements` cannot have the same name.
 - `disj`, line 8, prevents the same `Attribute` from being declared in different `Classes`.
 - the signature fact, line 12, enforces the parent relation to be acyclic, that is, no `Class` can be its own ancestor.
 - the signature fact, line 13, prevents the same `Attribute` from being declared in both a `Class` and an `AssociationClass`.
 - the signature fact, line 18, enforces each `Attribute` to be either declared in a `Class` or in an `AssociationClass` (hence preventing the presence of orphan `Attributes`).
 - `disj`, line 24, prevents two `AssociationClasses` from being linked to the same `Association`.
 - `disj`, line 25, prevents the same `Attribute` from being declared in different `AssociationClasses`.

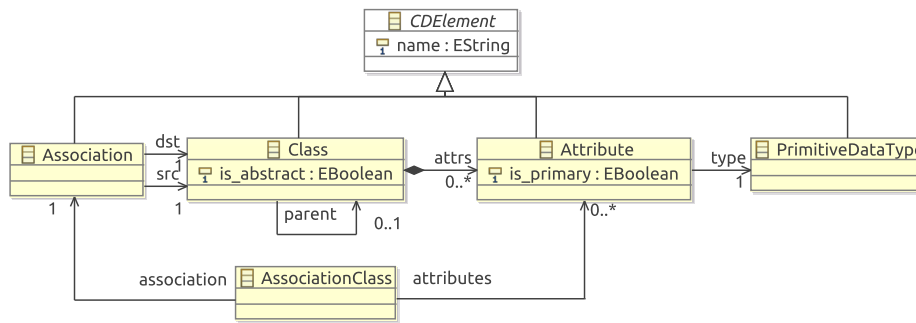


Figure 3.2: CD metamodel (adapted from [2])

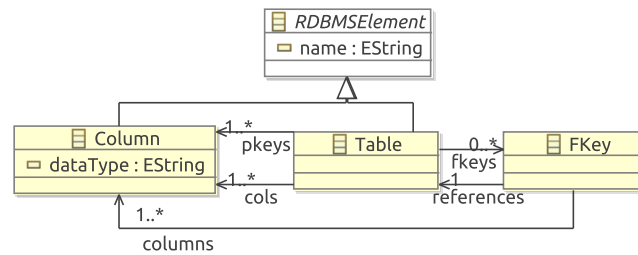


Figure 3.3: RDBMS metamodel (adapted from [2])

```

1 module CD
2 open util/boolean
3
4 abstract sig CDElement{
5   name: disj String
6 }
7 sig Class extends CDElement{
8   attrs : disj set Attribute,
9   parent: lone Class,
10  is_abstract: Bool
11 }{
12   this not in this.^@parent
13   attrs & AssociationClass.attributes =none
14 }
15 sig Attribute extends CDElement{
16   is_primary: Bool,
17   type: PrimitiveDataType
18 }{ this in Class.attrs+AssociationClass.attributes
19 }
19 sig Association extends CDElement{
20   src : Class,
21   dest : Class
22 }
23 sig AssociationClass{
24   association: disj Association,
25   attributes: disj set Attribute
26 }{attributes.is_primary=False}
27 sig PrimitiveDataType extends CDElement{}{
28   PrimitiveDataType.@name= "String"+"int"
29 }

```

Listing 3.1: CD Alloy module

```

1 module RDBMS
2
3 abstract sig RDBMSElement{
4   name: disj seq String
5 }
6 sig Table extends RDBMSElement{
7   cols: disj some Column,
8   pkeys: some Column,
9   fkeys: set FKey
10 }{
11   pkeys in cols
12 }
13 sig Column extends RDBMSElement{
14   dataType: String
15 }{
16   this in Table.cols
17   dataType in "TEXT"+"NUMBER"
18 }
19 sig FKey{
20   references: Table,
21   columns: some Column
22 }{
23   this in Table.fkeys
24   columns in this.~fkeys.cols
25 }

```

Listing 3.2: RDBMS Alloy module

- the signature fact, line 26, prevents an `Attribute` declared in an `AssociationClass` from being primary.
 - the signature fact, line 28, coupled with the constraint on line 5, enforces the presence of exactly two `PrimitiveDataType` named “String” and “Int”.
- In the RDBMS module,
 - `disj`, line 4, enforces that different `RDBMSElements` cannot have the same name.
 - `disj`, line 7, prevents the same `Column` from being contained in different `Tables`.
 - the signature fact, line 11, enforces columns composing the primary key (`pkeys`) of a `Table` to be a subset of the `Columns` contained in the said table.
 - the signature fact, line 16, enforces each `Column` to be contained in a `Table` (hence preventing the presence of orphan `Column`).
 - the signature fact, line 17, enforces the type of any `Column` to be either “TEXT” or “NUMBER”.
 - the signature fact, line 23, enforces each `FKKey` to be contained in the set of foreign key of a `Table` (hence preventing the presence of orphan `FKKey`).
 - the signature fact, line 24, enforces columns composing an `FKKey` to be contained in the `Table` to the foreign key belongs to.

3.2.2 CD2RDBMS: A Class Diagram to Relational Database Management System Model Transformation

The CD2RDBMS model transformation is the defacto standard case study when it comes to benchmarking a model transformation language [2].

Informal Specification

The source and target metamodels of this model transformation (CD and RDBMS, respectively) are shown as UML class diagrams in Fig. 3.2 and 3.3, respectively. Well-formedness constraints of those metamodels can be found in their respective Alloy representations, given in Listing 3.1 and 3.2.

We now give an informal specification of this transformation.

For each class c without a parent, a table is created. This table is populated with columns (1) representing the attributes of c or of its inheriting classes, (2) issued from associations having c as their source.

In case (1), the column is typed and named after the represented attribute. If the class declaring the attribute has a parent, names of all the parents of the declaring class have to appear in the name of the representing column.

In case (2), a column is created for each primary key of the table representing the class at the destination of the association, and is named after the association and the attribute it represents.

To better grasp the expected behavior of the transformation, we provide a visualization of a CD2RDBMS application in Fig. 3.4. We invite the reader to pay particular attention to the traceability links (dashed arrows). In this figure we see that two tables are created from the two topmost classes A and B. The columns `a` and `c_c` of table A and

the column `b1` of table `B` are obtained as per rule (1). The column `x_b1` composes the foreign key of table `A`, representing association `x` and referring to table `B`, and is obtained as per rule(2).

TGG implementation

TGG [80] (Triple Graph Grammar) is a well-known formalism allowing the declarative specification of model transformations. TGG specifications are composed of rules, each of them being expressed using three graphs. The source graph represents the subgraph whose match will trigger the rule. The target graph represents the subgraph which will be generated when the rule is triggered. A third graph called correspondence graph keeps track of relations between source and target elements for future reference. In this work, we reuse the partial¹ TGG implementation of the `CD2RDBMS` provided in [4]. We prefer this solution over others for the concise overview of the `CD2RDBMS` transformation it provides.

Figure 3.5 provides an overview of the solution and of the kind of traces (nodes of the correspondence graph) used to relate elements of `CD` (source graph) to elements of `RDBMS` (target graph).

The rules composing the transformation are given in Fig. 3.6. We note that those rules are bi-directional, *i.e.*, they can be read either from left to right or right to left. We are only interested in going from `CD` to `RDBMS` (left to right) and thus provide a reading in that direction solely:

- `C2T` enforces, via the `CT` correspondence node, the creation of a table for every class so that table and related class share the same name.
- `SC2T` enforces inheriting classes to be related to the same table as their parents (via `CT` as well).
- `PA2C` enforces the creation of a primary column for each primary attribute of a class. The column is named and typed after the attribute it has been created from and is contained in the table related to the class declaring the attribute.
- `A2FK` enforces the creation of an `FK` key for each association and of a column for each primary column of tables related to associations' destination classes. The created columns are typed after those primary key columns' type and are named after those columns and the association that led to their creation. Those created columns are also composing the foreign key created from the association and referring to the table representing its destination class.

F-Alloy Implementation

In Listing 3.3, we give an F-Alloy specification of the `CD2RDBMS` transformation. This F-Alloy specification is more concise than the equivalent Alloy specification presented in Annexe A.

The `CD2RDBMS` module (declared on line 1) from `CD` (imported on line 2) to `RDBMS` (imported on line 3) is composed of four `CREATE` mappings (lines 6 to 9). `CREATE` mappings are used to model the creation of elements in the output of the transformation. A mapping declaration consists of a sequence of arrow-separated signatures, the last one being the type of elements the mapping can produce. The pre and post conditions of a mapping are formulated in the *guard* and *value* predicates.

¹Persistence is abstracted away and columns name does not reflect inheritance

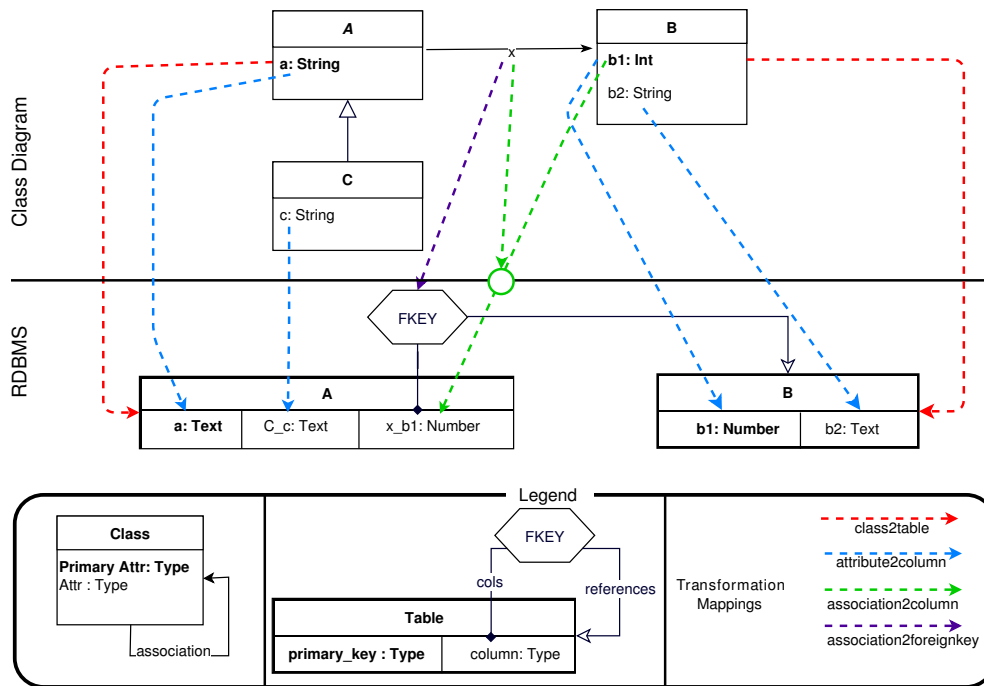


Figure 3.4: Execution of the CD2RDBMS transformation

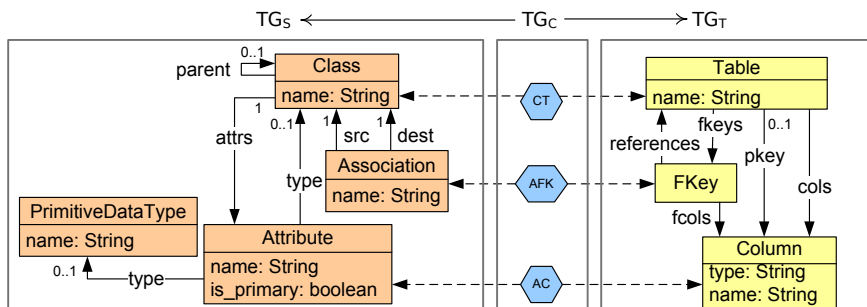


Figure 3.5: A simplified overview of the CD2RDBMS TGG transformation as given in [4]

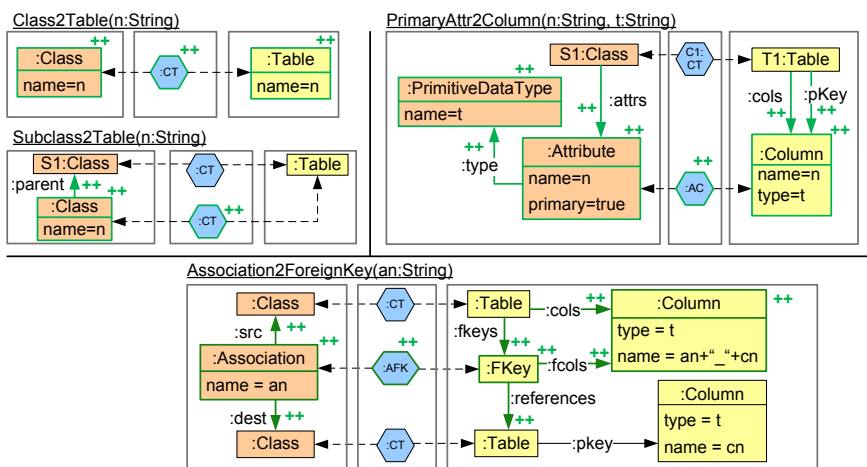


Figure 3.6: TGG Rules composing the CD2RDBMS transformation proposed in [4])

3.2. CASE STUDIES

```

1 module CD2RDBMS
2 open CD
3 open RDBMS
4
5 one sig CREATE{
6   class2table: Class -> Table,
7   attribute2column: Attribute -> Column,
8   association2column: Association ->
9     Attribute -> Column,
10  association2FKey: Association -> FKey,
11 }
12 pred guard_class2table(c:Class){
13   c.parent=none
14 }
15 pred value_class2table(c:Class , t:Table){
16   t.name[0]=c.name
17 }
18
19 pred guard_attribute2column(a:Attribute){
20   a not in AssociationClass.attributes
21 }
22 pred value_attribute2column(a:Attribute , c:
23   Column){
24   c.dataType=(a.type.name="String" implies "
25     TEXT" else "NUMBER")
26   c.name[0]= a.name
27   c.name[1]=(a.~attrs.parent)!=none implies a
28     .~attrs.name else none)
29 }
30
31 all i:Int | i>=1 and i< #(a.~attrs.^parent)
32   implies c.name[add[1,1]]= c.name[i].~name.
33   parent.name
34
35 a.is_primary=True implies c in CREATE.
36   class2table[a.~attrs.*parent].pkeys
37
38 c in CREATE.class2table[a.~attrs.*parent].
39   cols
40 }
41
42 pred guard_association2column(ass:Association,
43   att:Attribute){
44   att.is_primary=True and att in ass.dest.
45   attrs
46 }
47
48 pred value_association2column(ass:Association
49   , att:Attribute, c:Column){
50   c.dataType=(att.type.name="String" implies "
51     TEXT" else "NUMBER")
52   c.name[0]=ass.name
53   c.name[1]=att.name
54   c in CREATE.class2table[ass.src].cols
55 }
56
57 pred guard_association2FKey(a:Association){
58   value_association2FKey(a:Association , f:
59     FKey){
60     f.references=CREATE.class2table[a.dest]
61     f.columns=CREATE.association2column[a,
62       Attribute]
63     f in CREATE.class2table[a.src].fkeys
64 }
65 }

```

Listing 3.3: F-Alloy specification of the CD2RDBMS transformation

We draw a parallel between the given F-Alloy specification (mappings with their respective pre and postconditions) and the previously listed TGG rules in the following:

1. `class2table`: enforces the creation of a `Table` for each top-most `Class` (line 13) so that the `Table` is named after the `Class` (line 16). This mapping covers the `C2T` rule defined in TGG.
2. `attribute2Column`: models the creation of a `Column` for each `Attribute` which is not declared in an `AssociationClass`(line 20). The `Column` is typed (line 23) and named (line 24) after the `Attribute`. The name of the `Column` also contains the name of the `Classes` in the inheritance hierarchy of the `Class` declaring the `Attribute` if this latter has a parent (lines 25 and 26). Note that this requirement of including the name of inherited classes in the column's name is lacking in the provided TGG implementation. The `Column` composes the set of columns of the `Table` representing the top-most `Class` affiliated to the `Attribute` (line 28), and is also in the set of primary keys of the said `Table` if the `Attribute` is primary (line 27). This mapping covers the `PA2C` rule defined in TGG.
3. `association2column`: enforces the creation of a `Column` for each primary `Attribute` of a given `Association`'s target `Class` (line 32). The `Column` is typed after the `Attribute` (line 35) and named after both the `Association` (line 36) and the `Attribute` (line 37). The `Column` has to be in the set of columns of the `Table` representing the `Class` at the source of the `Association` (line 38). The `association2column` mapping covers the `Column` creations of the `A2FK` rule.
4. `association2FKey`: enforces the creation of an `FKey` for each `Association` (line 41). The `FKey` references the `Table` representing the `Association`'s target `Class`

(line 43). It is composed of the column mapped to the represented association by the `association2column` mapping (line 44) and is in the set of foreign keys of the `Table` associated via the `class2table` mapping to the class at the source of the `Association` (line 45). The `association2FKKey` mapping covers the `FKKey` creations of the `A2FK` rule.

We note that `SC2T` is not represented as a mapping in F-Alloy as it is not used to create any element in the output. The use of `SC2T` is to relate inheriting classes to the table representing their topmost ancestors. In (F-)Alloy we use the expression `c.*parent` (where `c` is an expression of type `class`) to return the set of all parents of `c`. The expression `Create.class2table[c.*parent]` then returns the table associated to the ancestors of `c`, just as defined by `SC2T`.

Figure 3.4 shows traces corresponding to each of the aforementioned mappings. In this figure, we can see amongst other details that tables `A` and `B` are created from classes `A` and `B` via the `class2table` mapping, respectively. No table `C` is created from class `C`, as it inherits class `A`. Column `a`, `b1`, `b2`, and `C_c` are all created from the `attribute2column` mapping. Column `x_b1` composing the foreign key created by the `association2FKKey` mapping is created by the `association2column` mapping.

3.2.3 CDRefinement: A Class Diagram Refinement Scenario

The `CDRefinement` endogenous model transformation is the specification of a refinement transformation applied to class diagrams with association classes. We choose this case study because it exercises the three elemental operations commonly used in endogenous model transformation, i.e., creation, deletion and update of elements (see [57]).

Informal Specification

The `CDRefinement` transformation consists in:

1. replacing all association classes by regular classes,
2. turning non-inherited abstract classes into non-abstract classes.

The purpose of (1) is to adapt the source model to languages in which association classes cannot be represented (*e.g.* `Ecore`), while (2) adapts the source model to languages where non-inherited abstract classes are not supported as such (*e.g.*, in Alloy, non-inherited abstract classes are considered non-abstract¹).

The manipulations to achieve (1) consist (as described in [81]) in removing each association class and its associated association, and in creating a new class containing the same attributes than the removed association class as well as two new associations, both connecting the originally associated classes to the newly created class. To achieve (2) it is sufficient to update the `abstract` modifier of concerned classes.

Henshin Implementation

We have chosen to use the graph based model transformation language Henshin [82] over other languages for its popularity and intuitive graphical syntax. Henshin's graphical

¹<http://stackoverflow.com/questions/27335623/sig-is-abstract-but-alloy-analyzer-make-an-instance-of-it>

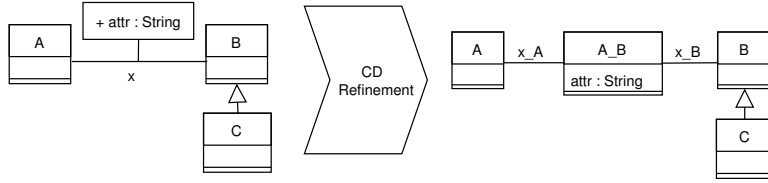


Figure 3.7: Execution of the CDRefinement transformation

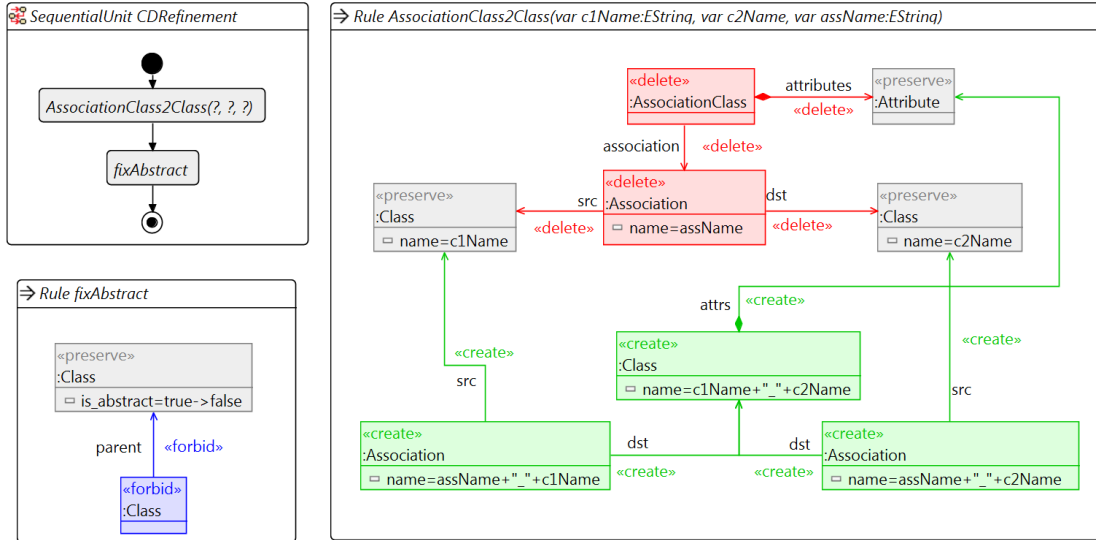


Figure 3.8: CDRefinement transformation defined using Henshin

syntax, as opposed to TGG’s one, enables the concise depiction of in-place operations, notably through the use of colors and labels.

A possible Henshin implementation of our case study is given in Fig. 3.8.

Two rules and a *sequentialUnit* defining in which order the rules are to be executed are depicted in Fig. 3.8. The first rule, namely *AssociationClass2Class*, defines that every *AssociationClass* and connected association are to be deleted. It also enforces that for each such deletion, a new class is created, containing those attributes declared in the deleted *AssociationClass*, and named after the two previously associated classes. Two associations are created as well, each connecting one of the previously associated classes to the newly created class. Those associations are named after the deleted association and the classes they are associating to the new class.

The second rule, namely *fixAbstract*, simply changes the value of *is_abstract* from true to false, for each class that has no children.

F-Alloy Implementation

We provide in Listing 3.3 a possible F-Alloy specification of the *CDRefinement* model transformation. This F-Alloy specification is more concise than the equivalent Alloy specification presented in Annexe B.

We explain it, basing ourselves on the Henshin implementation given earlier, as follows:

The *CD2Refinement* module (line 1) describes an endogenous transformation as it imports only one module (*CD*)(line 2). It is composed of the three signatures *CREATE* (line 4), *UPDATE* (line 26) and *DELETE* (line 37).

```

1 module CDRefinement
2 open CD
3
4 one sig CREATE{
5   associationClass2Class: AssociationClass ->
6     Class,
7   newAssociations: Class-> Association ->
8     Association
9 }
10 pred guard_associationClass2Class(a:
11   AssociationClass){
12   y.name= a.association.name
13   y.attrs=a.attributes
14   y.is_abstract=False
15   y.parent=none
16 }
17 pred guard_newAssociations(c:Class,a:Association)
18   c in a.(src+dest) and a.~association!=none
19 }
20 pred value_newAssociations(c:Class,a:Association,y:
21   Association){
22   y.name= a.name+c.name
23   y.src=(c=a.src implies c else CREATE.
24     associationClass2Class[a.~association])
25 }
26 one sig UPDATE{
27   fixAbstract: Class -> Class
28 }
29
30 pred guard_fixAbstract(c:Class){
31   c.is_abstract=True and c.~parent=none
32 }
33 pred value_fixAbstract(c:Class,y:Class){
34   y.is_abstract=False
35 }
36
37 one sig DELETE{
38   associationWithClass:Association,
39   associationClass :AssociationClass
40 }
41
42 pred guard_associationWithClass(a:Association){
43   a.~association!=none
44 }
45 pred guard_associationClass(a:AssociationClass){
46 }

```

Listing 3.4: F-Alloy specification of the CDRefinement transformation

The CREATE signature contains two mappings. The first, `associationClass2Class`, enforces the creation of a new `Class` for any `AssociationClass` present in the source model (line 9). This newly created `Class` is named (line 11) after the `AssociationClass` it represents and contains the same attributes (line 12). This newly created class is not abstract (line 13) and has no parents (line 14).

The second one, `newAssociations`, ensures the creation of an `Association` for each combination of `Class` and `Association` (c, a) present in the source model with a linked to c and adorned by an `AssociationClass` (line 18). The created `Association` is named “ a_c ” (line 21) and then linked to either c or the `Class` replacing the `AssociationClass` previously adorning a depending on whether the source association was pointing to or was coming from an `AssociationClass` (line 22 and 23).

The previously adorned associations as well as all the association classes are removed from the source model as specified by the two DELETE mappings (line 38 and 39).

All those aforementioned mappings express the Henshin rule `AssociationClass2Class`.

The Henshin rule `fixAbstract` is represented by the UPDATE mapping of the same name.

The UPDATE mapping `fixAbstract` simply enforces that each abstract class without children (line 31) should have their `is_abstract` field set to `False` (line 34). Other fields of `Class` remain unchanged (and are thus not assigned in the value predicate).

In this chapter, we present our work on adapting Alloy to the specification of functions so that they can be computed efficiently – i.e., a target instance can be obtained in polynomial time given an source instance – while retaining the possibility of using Alloy’s automatic analysis on those function for validation and verification’s sake.

The solution we propose to solve this problem takes the form of a new language, named F-Alloy.

F-Alloy’s syntax is a restriction of the Alloy syntax which, in the context of a model transformation specification and given a source instance, can be imperatively processed to obtain an instance of the transformation specification. Though F-Alloy specifications

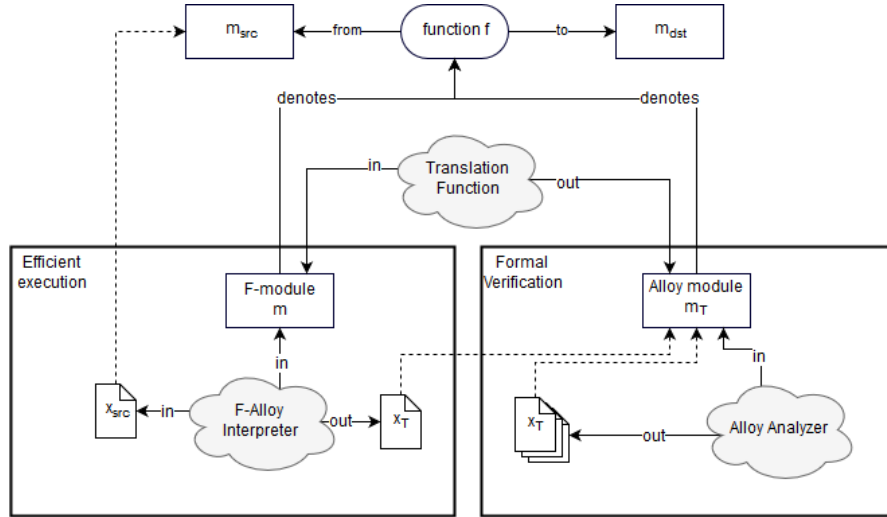


Figure 3.9: An overview of the F-Alloy approach

are syntactically valid Alloy specification, the semantics of F-Alloy differ from Alloy in the sense that redundant constraints specific to model transformation expressed in Alloy (previously identified as `C_PRE`, `C_POST`, `C_TRACE` in Section 2.4) remains implicit in F-Alloy, hence enabling a more concise formulation of transformations.

In short, F-Alloy is designed to have the following properties:

- F-Alloy is a domain specific language specifically designed to define model transformations. An F-Alloy model transformation specification is called an *f-module*.
- F-Alloy’s syntax is a restriction of the Alloy syntax in the sense that any f-module is a syntactically valid Alloy module
- F-modules can be efficiently computed (by a process we call *interpretation*)
- The semantics of F-Alloy is given as a translation to Alloy, thus allowing f-modules to be subject to Alloy analysis.

The way F-Alloy integrates with Alloy is depicted in Fig. 3.9. In this figure, we are interested in specifying a function f from a source to a target metamodel. To do so, we produce an f-module “denoting” f . This f-module can be interpreted, given a valid a_{src} -instance x_{src} , to efficiently build the instance x_f corresponding to the application of f on x_{src} . The f-module can also be analyzed using Alloy’s analysis by first translating it to an Alloy module denoting the same transformation.

3.3 Syntax of F-Alloy

This section aims at providing a formal yet comprehensive introduction to the F-Alloy syntax. We first define the syntax of F-Alloy reusing Alloy’s syntactic constructs. We then give the purpose of each F-Alloy syntactic construct defined and finish by showing that those constructs are all syntactically valid Alloy constructs.

3.3.1 F-Alloy’s Syntax

To avoid confusion and for this work to be self contained, we reuse the BNF notation used in [24] to define Alloy’s syntax. This notation is introduced as follows:

“The grammar uses the standard BNF operators:

- x^* for zero or more repetitions of x ;
- x^+ for one or more repetitions of x ;
- $x|y$ for a choice of x or y ;
- $[x]$ for an optional x .

In addition,

- $x,^*$ means zero or more comma-separated occurrences of x ;
- $x,^+$ means one or more comma-separated occurrences of x ;

To avoid confusion, potentially ambiguous symbols – namely parentheses, square brackets, star, plus and the vertical bar – are set in bold type when they are to be interpreted as terminals rather than as meta symbols. The string `name` represents an identifier and `number` represents a numeric constant, ...” [24].

The F-Alloy BNF is given in Listing 3.6. It reuses syntactic constructs of the Alloy BNF given in Listing 3.5.

We split the BNF definition of F-Alloy into two parts in order to ease its understanding. While the first part reveals the structure of *f-modules* – i.e., modules expressed in F-Alloy – the second part defines the subset of boolean-valued Alloy expressions, called *rules*. We bring the reader’s attention to the fact that the syntactic constructs `expr`, `name`, `qualName`, `decls`, `moduleDecl`, and `import` present in the BNF are coming from the Alloy BNF given in Listing 3.5.

The presented F-Alloy BNF allows the expression of both endogenous and exogenous specifications. We note the presence of `UPDATE` and `DELETE` signatures, which together with `CREATE` are used to define endogenous transformation following the approach presented in Section 2.4.1.

The nature of an f-module specification is determined by the number of open statements it contains (one for endogenous, two for exogenous) and by the presence or absence of the `UPDATE` and `DELETE` signatures (those are only allowed in endogenous specifications). We say that an f-module from a_{src} to a_{dst} defines a transformation from a_{src} to a_{dst} and introduce the following notation.

Notation 2 (F-module). *An f-module f from a_{src} to a_{dst} is denoted by $f : a_{src} \rightarrow a_{dst}$ or more concisely, f if the domain and range of the function it defines are known or irrelevant.*

In this thesis, we use f to range¹ over f-modules and a to range over Alloy modules.

3.3.2 A Formal Definition of Mappings

Before illustrating the usage of the previously introduced F-Alloy’s syntax, we provide a formal definition to mappings and their surrounding concepts as we often refer to those in the remaining sections.

¹non exclusively: f can also used to denote functions.

3.3. SYNTAX OF F-ALLOY

```

1 alloyModule ::= [moduleDecl] import* paragraph*
2 moduleDecl ::= module qualName [[ name,+]]
3 import ::= open qualName [[ qualName,+]] [ as name]
4 paragraph ::= sigDecl | factDecl | predDecl | funDecl | assertDecl | cmdDecl
5 sigDecl ::= [abstract] [mult] sig name,+ [sigExt] { decl,*} [block]
6 sigExt ::= extends qualName | in qualName [+qualName]*
7 mult ::= lone | some | one
8 decl ::= [disj] name,+ : [disj] expr
9 factDecl ::= fact [name] block
10 predDecl ::= pred [qualName.]name [paraDecls] block
11 funDecl ::= fun [qualName.]name [paraDecls] : expr {expr}
12 paraDecls ::= ( decl,* ) [[ decl,*]
13 assertDecl ::= assert [name] block
14 cmdDecl ::= [name :] [run|check] [qualName|block] [scope]
15 scope ::= for number [ but typescope,+ ] | for typescope,+
16 typescope ::= [exactly] number qualName
17 expr ::= const | qualName | @name | this | unOp expr | expr binOp expr
18         | expr arrowOp expr | expr [expr,*] | expr [!|not] compareOp expr
19         | let letDecl,+ blockOrBar | quant decl ,+ blockOrBar
20         | {decl,+ blockOrBar} | (expr) | block
21 const ::= [-] number | none | univ | iden|
22 unOp ::= ! | not | no | mult | set | # | ~ | * | ^
23 binOp ::= || | or | && | and | <=> | iff | => | implies | & | + | - | ++ | <: | :> | .
24 arrowOp ::= [mult|set] -> [mult|set]
25 compareOp ::= in | = | < | > | =< | >=
26 letDecl ::= name = expr
27 block ::= {expr*}
28 blockOrBar ::= block | bar expr
29 bar ::= |
30 quant ::= all | no | sum | mult
31 qualName ::= [this/] (name/)* name

```

Listing 3.5: Alloy BNF as given in [24]

```

1 fmodule ::= moduleDecl import [import] fparagraph*
2 fparagraph ::= fsigDecl | guardDecl | valueDecl
3 fsigDecl ::= one sig sigName { mappingDecl,*}
4 sigName ::= CREATE | UPDATE | DELETE
5 mappingDecl ::= name : qualName (->qualName)*,
6 guardDecl ::= pred guard_name [paraDecls] block
7 valueDecl ::= pred value_name [paraDecls] rBlock
8 rBlock ::= {rule*}
9
10 rule ::= strict | loose | step | conditional
11 conditional ::= expr implies rule
12 strict ::= name.name [[expr]] = expr
13 loose ::= name in image.name [[expr]]
14 image ::= sigName.name [expr]
15 step ::= all i: Int | range implies name.name [add[i,1]] = expr
16 range ::= i (>|>=) expr and i (<|<=) expr

```

Listing 3.6: F-Alloy BNF

Definition 5 (Mappings, domain, range). *Considering an f-module $f : a_{src} \rightarrow a_{dst}$, we call the fields declared in f relating signatures of a_{src} to signatures of a_{dst} mappings. For a mapping $map: X_1 \rightarrow \dots \rightarrow X_n \rightarrow Y$ the domain denotes the tuple of signatures (X_1, \dots, X_n) and the range denotes the signature Y . Mappings declared in a `DELETE` signature being of the form $map: X$, they do not have range.*

To illustrate this definition, let us consider the following mappings

- from the `CD2RDBMS` transformation given in Listing 3.3:

```
— association2column : Association -> Attribute -> Column
```

The domain of the create mapping `association2column` is the tuple of signature `(Association, Attribute)` and the range is the signature `Column`

- from the `CDRefinement` transformation given in Listing 3.4:

```
— associationClass2Class : AssociationClass -> Class
```

The domain of create mapping `associationClass2Class` is the signature `AssociationClass` and the range is the signature `Class`

```
— fixAbstract : Class -> Class
```

The domain and range of update mapping `fixAbstract` is the signature `Class`.

```
— associationClass : AssociationClass
```

The domain of the delete mapping `associationClass` is the signature `AssociationClass`. This mapping being declared in the `DELETE` signature, it does not have any range.

3.3.3 F-Alloy's Syntax Usage

To help the reader understand why F-Alloy's syntax is defined the way it is, and to give a short introduction on how the syntax is used, we list in the following itemization the intentions behind each concept declared in Listing 3.6, and exemplify their usage using excerpts of the `CD2RDBMS` and `CDRefinement` case studies given in Listing 3.3 and 3.4:

- `f-module`: Any specification written in F-Alloy is called an f-module and consists of an Alloy module declaration (`moduleDecl`)(identifying the module), some `import` declarations corresponding to the transformation's source and target modules and of a body composed of several `fparagraph`.

Example: The `CDR2RDBMS` specification given in Listing 3.3 is a syntactically valid `f-module`

- `fparagraph`: A paragraph in F-Alloy takes either the form of a signature declaration (`fsigDecl`), a guard declaration (`guardDecl`) or a value declaration (`valueDecl`). Guards and values are both Alloy predicates.

Example: The `CDR2RDBMS` specification given in Listing 3.3 is composed of 9 `fparagraph` (a `CREATE` signature, four guard and four value predicates)

- `fsigDecl`: In F-Alloy, signatures are declared as singleton (`one`) and have as sole purpose to act as container for the mappings (`mappingDecl`) composing the transformation. In exogenous transformations, only one signature, named `CREATE` is allowed. In endogenous transformations, there are three signatures named `CREATE`, `UPDATE` and `DELETE` after the different kind of operations used in endogenous transformations:

- CREATE mappings are used to express the creation of atoms typed by their range for given tuples typed by their domain.
- UPDATE mappings have their domain and range bound to be the same signature. They are used to express a substitution: if an atom a is mapped via an update mapping to an atom b , then a will be replaced by b . All the tuples referring to a will then refer to b instead.
- DELETE mappings do not have a range and are used to express that an element is to be deleted from an instance.

Example: The following excerpt from Listing 3.3 consists of a signature declaration and four mapping declarations.

```
one sig CREATE{
  class2table: Class -> Table,
  attribute2column: Attribute -> Column,
  association2column: Association -> Attribute -> Column,
  association2FKey: Association -> FKey,
}
```

- guardDecl: A guard predicate contains the precondition under which a mapping is to be triggered. The sequence of parameters it takes (`paraDecls`) corresponds to the domain of the mapping it is associated to. The association between mapping and guard predicate is done by name, *i.e.*, the guard of a mapping called g will be named “guard_ g ”.

Example: The following excerpt from Listing 3.3 shows the guard predicate of the `association2column` mapping. We can see that the types of the declared parameters `ass` and `att` match the type of the signatures in the domain of the `association2column` mapping.

```
pred guard_association2column(ass:Association, att:Attribute){
  att.is_primary= True and att in ass.dest.attrs
}
```

This guard states that the `association2column` mapping will be triggered for any pair of association and attribute such that the attribute is primary and such that association’s destination is the class in which the attribute is declared.

- valueDecl: A value predicate contains a set of rules (`rule`) defining the values of fields of the elements created by the mapping it is associated with. Association between mapping and value predicate is done by name, just like for guard predicate. The sequence of parameters (`paraDecls`) a value predicate takes corresponds to the domain and range of the associated mapping.

Example: The following excerpt from Listing 3.3 shows the value predicate associated to the `class2table` mapping. Parameters `c` and `t` are typed after the domain and range of the `class2table` mapping.

```
pred value_class2table(c:Class , t:Table){
  t.name[0]= c.name
}
```

This value predicate simply enforces that any table created through the `class2table` mapping should be named after its associated class.

- rule: Rules in F-Alloy are Alloy expressions that restrict the value of fields in the target module.

- `conditional` rules enable each rule to be preceded by a condition with effect that the rule is applied if and only if the condition is satisfied.

Example: a loose rule here is to be applied if and only if the attribute `a` given as parameter is primary.

```
a.is_primary= True implies c in CREATE.class2table[a.~attrs.*parent].pkeys
```

- `strict` rules are direct restrictions of the value of a created or updated atom's field.

Example: We restrict the name of a table t^1 to be the name of the class c .

```
t.name[0]= c.name
```

- `loose` rules are restrictions on the value of fields of an atom created or updated by another mapping.

Example: The created column c composes the set of `pkeys` of the table created, through the `class2table` mapping, from the parents of the class declaring a (or the class itself if its has no parent).

```
c in CREATE.class2table[a.~attrs.*parent].pkeys
```

- `step` rules are used to inductively restrict a field of a created or updated atom to be composed of certain tuples.

Example: each entry in the name of the column c corresponds to the name of the parent of the previous entry, with the first entry (index 0) being already defined through the use of a strict rule.

```
all i:Int| i>=1 and i< #(a.~attrs.^parent) implies  
c.name[add[i,1]]= c.name[i].~name.parent.name
```

For completeness sake, we now give well-formedness constraints for the BNF given in Listing 3.6.

3.3.4 F-Alloy's Well-Formedness Constraints

Let us consider an f-module defining a transformation from a_{src} to a_{dst} . That f-module is syntactically valid if the following constraints hold.

- Constraints ruling F-Alloy's overall structure:

ImportWF In the `fmodule` rule, there is exactly one `import` occurrence in the case the f-module defines an endogenous transformation and exactly two `import` occurrences in the case the f-module defines an exogenous transformation. In these `import` rules, `qualName` should refer to an Alloy module. In the case of an exogenous transformation, the two referred Alloy modules should be distinct. Those modules are named *source module* and *target module*, respectively (as they represent the source and target metamodells of the specified transformation).

SigWF There is exactly one signature declaration (produced by `fSigDecl`) with name `CREATE` in an f-module defining an exogenous transformation, and exactly three signature declarations (produced by `fSigDecl`) with name `CREATE`, `UPDATE` and `DELETE` in an f-module defining an endogenous transformation.

¹at index 0 as name is declared as a sequence of string

GuardWF For each mapping declaration produced by the `mappingDecl` rule, there exists exactly one guard declaration (produced by the `guardDecl` rule) whose name is the name of the mapping prefixed by “guard_”. The guard declaration defines a predicate describing what should hold in the source instance in order for the mapping to take effect. The parameters of the guard predicate should be typed by the domain of the mapping. In a guard’s `block`, `expr` occurrences are boolean valued Alloy expressions that may solely contain features of a_{src} and input parameters of the enclosing predicate, guards having as role to define patterns to be matched in the input instance. A guard predicate is the F-Alloy equivalent of QVT’s `checkonly` pattern.

ValuedWF Similarly, for each mapping declaration produced by the `mappingDecl` rule (except for those present in a `DELETE` signature), there exists exactly one value declaration (produced by the `valueDecl` rule), whose name is the name of the mapping prefixed by “value_”. The value declaration defines a predicate describing what should be in the target instance after applying the mapping. The parameters of the value predicate should be typed by the domain and range of the mapping. A value predicate is the F-Alloy equivalent of QVT’s `enforce` patterns.

CreateWF In the `CREATE` signature, `mappings` denotes n-ary relation. The last occurrence of `qualName` should refer to a signature defined in the target module, while other occurrences of `qualName` refer to signatures defined in the source module. There is one guard and one value predicate associated to each create mapping with effect that atoms in the source instance satisfying the guard predicate will lead to the creation of an atom typed by the range of the mapping whose value is defined by the value predicate.

UpdateWF In the `UPDATE` signature, mappings are binary relations having the same signature as domain and range. They are also associated to one guard and one value predicate each. The intention is to relate each atom typed by the domain of the mapping and satisfying the guard to a new output atom of same type to express that the former is to be replaced by the latter. The values of the fields of the updated atom are defined by the value predicate. Unassigned fields are left unchanged.

DeleteWF In the `DELETE` signature, “mappings” are unary relations that represent atoms to be deleted. They are hence solely associated to a guard predicate with the intention that each atom typed by the domain of the mapping and satisfying the guard, as well as all references to that atom, are removed from the instance.

- Constraints governing F-Alloy’s rules:

ExprWF `expr` is an Alloy expression that may contain features of a_{src} and input parameters of the enclosing predicate as well as occurrences of `image`. This restriction allows to enforce that the value of fields in the target instances solely depends on the source instance (as we will prove in Section 3.4.3).

StrictWF In `strict` rules, the first `name` occurrence is the name of the output parameter (i.e., typed after the range of the mapping), and the second is the name of a field declared in the signature typing that parameter. The `expr` in square brackets is to appear if and only if this field has an arity greater than 2.

LooseWF In `loose` rules, the first `name` is the name of the output parameter and the second `name` is a field of the signature typing the `image`. The `expr` in square brackets is to appear if and only if this field has an arity greater than 2.

StepWF `step` rules are used to express inductive assignments. They are thus always preceded by a `strict` rule, composing the base of the induction, and featuring an integer-valued index. In `step`, the first `name` occurrence is the name of the output parameter and the second is the name of the field assigned in the preceding `strict` rule. The right-hand side – `expr` – of `step` should then contain an occurrence of `name.name[i]`.

RangeWF In `range`, the two `expr` are integer-valued and the first `expr` value is bound to be equal to the `index` value of the `strict` rule composing the base of the induction for well-formedness sake.

3.3.5 F-Alloy to Alloy Correspondences

We finish this introduction to the F-Alloy syntax by showing, based on the given Alloy and F-Alloy BNFs, that any F-Alloy syntactic construct is also a valid Alloy syntactic construct (in other words an f-module is a syntactically valid Alloy module).

Correspondences between F-Alloy’s syntactic constructs and their Alloy counterparts are detailed in the following itemization:

- `range` is a syntactically valid `expr` of the form `expr compareOp expr binOp expr compareOp expr`
- `step` is a syntactically valid `expr` of the form `quant decl bar expr binOp expr compareOp expr`
- `image` is a syntactically valid `expr` of the form `expr binOp expr`
- `loose` is a syntactically valid `expr` of the form `expr compareOp expr binOp expr`
- `strict` is a syntactically valid `expr` of the form `expr compareOp expr`
- `strict` is a syntactically valid `expr` of the form `expr compareOp expr`
- `conditional` is a syntactically valid `expr` of the form `expr binOp expr`
- `rule` is a syntactically valid `expr`
- `rBlock` is a syntactically valid `block`
- `valueDecl` is a syntactically valid `predDecl`
- `guardDecl` is a syntactically valid `predDecl`
- `mapDecl` is a syntactically valid `decl` taking into account that `expr -> expr` is itself a syntactically valid `expr`
- `sigName` is a syntactically valid `name`
- `fsigDecl` is a syntactically valid `sigDecl`
- `fparagraph` is a syntactically valid `paragraph`
- `fmodule` is thus a syntactically valid `alloyModule`

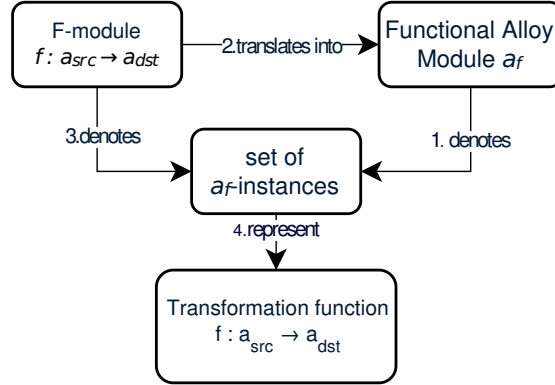


Figure 3.10: Relation between Alloy and F-Alloy’s semantics

3.4 Translational Semantics of F-Alloy

3.4.1 Overview

To ease the reading of this section, we picture the relation between Alloy and F-Alloy’s semantics in Fig. 3.10.

The semantics of Alloy is given in [24] using a denotational approach – each syntactic construct being mapped to a function from instance to boolean value, with the effect of defining which properties are enforced in an instance by the construct in question. In short, an Alloy module denotes a set of instances – edge number 1 of Fig. 3.10.

One of the great advantages in reusing the syntax of Alloy in the definition of the F-Alloy transformation language is the ease of translation between F-Alloy specifications and Alloy. Indeed, f-modules are by essence valid Alloy modules. Yet the meaning we give to f-modules differs from their original Alloy meaning. This difference stems from our design choice of leaving out (for conciseness sake) redundant constraints, identified in Section 2.4.1, needed in Alloy to enforce a “functional behavior”.

In the next subsection we thus define a translation function $\mathcal{T} : \mathcal{F}alloy \rightarrow Alloy$, with $\mathcal{F}alloy$ and $Alloy$ being the set of all possible f-modules and Alloy modules, respectively, – corresponding to edge number 2 of Fig. 3.10 – and set the meaning of an f-module f to be that of the Alloy module $\mathcal{T}(f)$ (as defined in [24]).

In a nutshell, the f-module f denotes the same set of instances as the Alloy module $\mathcal{T}(f)$.

In Section 3.4.3, we will show that for any f-module f , $\mathcal{T}(f)$ is a functional Alloy module, hence showing that instances of $\mathcal{T}(f)$, and thus transitively f , represent indeed a transformation function – corresponding to edge number 4 of Fig. 3.10.

3.4.2 Translating F-Alloy to Alloy

In this subsection, we define the translational semantics of F-Alloy by defining a translation function \mathcal{T} mapping each f-module f , defining either an endogenous or an exogenous model transformation, to an Alloy Module a_f . We define then the meaning of f to be the one Alloy gives to a_f .

When considered as an Alloy module (we recall that f-modules are syntactically valid Alloy modules), an f-module f only defines relations from signatures of the source to signatures of the target module. Those relations provide enough structure to obtain transformation instances from analysis. Yet, the way elements are related is not constrained. Only some predicates are present to give information on how elements of the

source and target should relate to each other. Constraints enforcing the transformation behavior (identified in Section 2.4.1) as well as other expected functional assumptions such as disjointness of the source and target instance, necessary to the simplification of the interpretation process, should thus be made explicit as Alloy facts to convey the real intent of the F-Alloy specification.

We thus define \mathcal{T} by listing procedures defining how to systematically add those constraints to an f-module in order to obtain the Alloy module defining its meaning. We recall from definition 5 that for a mapping $\text{map}: X_1 \rightarrow \dots \rightarrow X_n \rightarrow Y$, the domain denotes the sequence of signatures (X_1, \dots, X_n) and the range denotes the signature Y . We introduce the term of *input tuple* and *output atom* here to denote the set of tuples typed after the domain of a mapping and satisfying its associated guard and those atoms typed after the range of a mapping and who have been mapped to input tuples through that mapping, respectively.

We now list the constraints to be added to an f-module f in order to obtain a_f :

- Constraints to be added to any f-module f
 - **Map Disjunction.** Mappings declared in `CREATE` and `UPDATE` signatures define partial functions which have disjoint ranges.

The intent is to ensure that for any rule, the triggering of a guard will lead to the creation of a new atom. This constraint enables, when interpreting f-modules (see Section 3.5), to consider mappings one at a time when creating output elements, hence simplifying the interpretation process.

Example (CD2RDBMS): columns representing attribute and association should be disjoint.

```
CREATE.attribute2column[Attribute] & CREATE.association2column[Association,Attribute] =
    none
```

Map Disjunction constraints can be generated following the present procedure:

```
LET a_dst= m.getImportedModule(2) //2nd open statement
IF a_dst= null THEN //endogenous case
    a_dst= m.getImportedModule(1)
FI
FOR EACH signature Y DECLARED IN a_dst DO
    LET M = FIND ALL mappings having Y as range IN m
    FOR EACH mapping map IN M DO
        WITH map: "map: X1 -> .. -> Xn -> Y"
        LET sig= signature in which map is declared
        FOR EACH mapping map2 IN M DO
            IF map != map2 THEN
                WRITE IN MapDisjunction fact
                    "map[X1,..,Xn] & map2[X1,..,Xn] = none"
                WROTE
            FI
        DONE
    DONE
DONE
```

- **Map Injectiveness.** Functions defined by `CREATE` and `UPDATE` mappings are injective. The intent is again to simplify the process of creating output elements when interpreting f-modules. This time, this constraint enables us to consider candidates to trigger a mapping one at a time, a new output element being created at any triggering.

Example (CD2RDBMS): distinct attributes should be mapped to distinct columns, at most one attribute being mapped to a given column.

```
all y: Column | lone CREATE.attribute2column.y
```

Map Injectiveness constraints can be generated following the present procedure:

```
LET M = FIND ALL CREATE and UPDATE mappings DECLARED IN m
FOR EACH mapping map IN M DO
  WITH map: "map: X1 -> .. -> Xn -> Y"
  LET sig= signature in which map is declared
  WRITE IN MapInjectiveness fact
    "all y:Y| lone sig.map.y"
  WROTE
DONE
```

- **Predicate Association.** For each CREATE and UPDATE mapping there is an output atom exactly for those tuples in the domain that satisfy the guard predicate . The values of fields of the output atom are defined in the value predicate. For a DELETE mapping there is a tuple of the form (d, s) where d is a single DELETE atom for each atom s in the domain of the mapping satisfying the guard predicate.

The intent is to ensure that guard and value predicates play their expected role of pre- and post-conditions for the mappings they are associated to, hence filling the role of the C_PRE and C_POST constraints identified in Section 2.4.1.

Example (CD2RDBMS): a column y is associated to an attribute x if and only if the guard predicate is satisfied for x . In that case, the value predicate has to hold for x and y as well.

```
all x : Attribute{
  (guard_attribute2column[x] and one CREATE.attribute2column[x] and
   value_attribute2column[x, CREATE.attribute2column[x]])
  or
  (not guard_attribute2column[x] and no CREATE.attribute2column[x])
}
```

Predicate Association constraints can be generated following the present procedure:

```
LET M = FIND ALL mappings DECLARED IN f
FOR EACH map IN M DO
  WITH map: "map: X1 -> .. -> Xn -> Y"
  LET sig= signature in which map is declared
  IF sig="CREATE" or sig="UPDATE" THEN
    WRITE IN PredicateAssociation fact
      "all x1:X1|..|all xn:Xn{"
      "( guard_map[x1,..,xn] and one sig.map[x1,..,xn] and value_map[x1,..,xn,sig.map[x1
      ,..,xn]] )"
      "or ( not guard_map[x1,..,xn] and no sig.map[x1,..,xn] )"
    WROTE
  ELSE IF sig="DELETE" THEN
    WRITE IN PredicateAssociation fact
      "all x1:X1 {"
      "(guard_map[x1] and x1 in DELETE.map"
      "or (not guard_map[x1] and x1 not in DELETE.map) )"
    WROTE
  FI
DONE
```

- **Minimal Assignment.** The values of the fields of output atoms are limited to those explicitly specified through rules, except for UPDATE mappings, in which case values of fields which are not specified through rules are equal to the values of the same fields prior to update.

The intent of this constraint is to provide an upper-bound to fields for which values were partially specified, *e.g.*, using “in” instead of strict equality. The extra clause of this constraint concerning UPDATE mappings is here to make endogenous transformations less verbose. The idea is to avoid rewriting existing values by limiting the transformation to the expression of what has to change.

Example (CD2RDBMS): The name of a column in the range of the `association2column` mapping is a sequence of strings whose size (obtained by using the Alloy operator #) is equal to the number of elements returned by all the expressions explicitly assigned through rules.

```
#c.name=add[#att.name,#ass.name]
```

Example (CDRefinement): `isAbstract` is the only field present in the `noUselessAbstract` value predicate. Constraints are thus added to force other fields to keep their original values.

```
pred value_noUselessAbstraction(c:Class,y:Class){
  y.isAbstract=False
  y.attrs=c.attrs //added constraint
  y.parent=c.parent //added constraint
  y.name=c.name //added constraint
}
```

Minimal Assignment constraints can be generated following the present procedure:

```
FOR EACH mapping map DECLARED IN f DO
  WITH map: "map:X -> .. -> Xn -> Y"
  LET sig= signature IN which map is declared
  LET y = last parameter of value_map
  FOR EACH field g DECLARED IN signature Y DO
    LET constraint=""
    FOR EACH rule r CONTAINING g IN value_map DO
      IF r NOT CONTAINING loose THEN
        IF constraint="" THEN
          constraint=COUNT[r]
        ELSE
          constraint="add["+COUNT[r]+","+constraint+]"
        FI
      FI
    DONE
  WRITE IN value_map
    "#y.g="+COUNT[r]
  WROTE
  LET flag=false
  FOR EACH mapping map2 IN f DO
    WITH map2: "map2: X -> .. -> Xn -> Y"
    IF map != map2 THEN
      FOR EACH rule r IN value_map2 DO
        IF r CONTAINS loose THEN
          flag=true
          WRITE IN value_map2
            "#f.g="+COUNT[r]
          WROTE
        FI
      DONE
    FI
  DONE
  IF sig="UPDATE" THEN
    IF NO rule r CONTAINS g IN any value predicate of f THEN
      WRITE IN value_map
        "y.g = x.g"
      WROTE
    FI
  ELSE IF flag=false
```

```

        WRITE IN value_map
            "y.g = none"
        WROTE
    FI
DONE
FUNCTION COUNT[rule r] RETURNS expr
    IF r IS strict THEN
        WITH r: "y.g[[expr1]]=expr2"
        IF expr1 = "" OR expr1 IN INT THEN
            RETURN "#expr2"
        ELSE
            RETURN "mult[#expr1,#expr2]"
        FI
    FI
    IF r IS step THEN
        WITH r: "all i:Int|range implies y.g[add[i,1]] = expr"
        WITH range: "i > expr1 and i < expr2"
        LET c="max[sub[#expr2,#expr1]+0]"
        RETURN "mult[#expr,c]"
    FI
    IF r IS conditional THEN
        WITH c: "expr1 implies r2"
        RETURN "not expr1 implies 0 else "+COUNT[r2]
    FI
    IF r IS loose THEN
        WITH r: "y in image.name[expr]"
        IF expr = "" OR expr IN INT THEN
            RETURN "#image"
        ELSE
            RETURN "mult[#expr,#image]"
        FI
    FI
END FUNCTION

```

- Constraints to be added to any f-module f defining an endogenous model transformation:

- **IO Disjunction.** The set of all atoms in the input tuples of CREATE, UPDATE, and DELETE mappings are disjoint with the set of all output atoms of CREATE and UPDATE mappings.

The intent is to clearly separate through constraints the source instance from the target instance hence filling the role of the C_TRACE_EN constraint identified in Section 2.4.1. We note that these constraints are only needed for endogenous transformations as the type of atoms in an exogenous are signatures being declared either in the source or in the target module of the transformation, hence naturally dissociating atoms belonging to the source instance from those in the target instance.

Example (CDRefinement): the set of classes which are output atoms in the range of the mapping associationClass2Class and the set of classes in input tuples in the domain of the mapping newAssociation should be disjoint.

```
no associationClass2Class[AssociationClass] & newAssociations.Association.Association
```

IO Disjunction constraints can be generated following the present procedure:

```

FOR EACH mapping map DECLARED IN f DO
    WITH map1: "map1: X1 -> .. -> Xn -> Y"
    LET sig1=signature in which map1 is declared
    FOR EACH mapping map2 DECLARED IN f DO
        WITH map2: "map2: A1 -> .. -> An -> B"
        LET sig2=signature in which map2 is declared
        FOR EACH i IN RANGE [1, n]

```

```

IF Ai = Y THEN
  WRITE IN IODisjunction fact
    no sig1.map1[X1,..,Xn] & sig2.map2[A1,..,Ai-1].Ai+1(..).An.B
  WROTE
FI
DONE
DONE
DONE

```

- **Constraints Framing** Transformation instances of an endogenous f-module are composed of a source instance and of tuples, typed by CREATE, UPDATE and DELETE mappings, embodying the operations to be performed (see Fig. 3.12). This become problematic when considering the constraints declared in the source module of the transformation. Indeed, when running an analysis on such an endogenous specification, those aforementioned constraints should be satisfied in the transformation instance yet this later contains source module elements other than those present in the source instance given as input. This particularity of endogenous transformation makes it likely that analysis fails to obtain all expected transformation instances. A solution to this problem is to add a context to those source module constraints, so as to apply them on the source and target instance, respectively (instead of the transformation instance). To do so, all facts of the source module are to be rewritten as predicates, taking as parameter the set of atoms they are to be applied on. Those predicates are then called in a fact in the transformation module, where the parameter given correspond to the set of atoms composing the source instance (returned by f_{in}) and the target instance (returned by f_{out}), respectively. Constraints Framing hence covers the role of C_TRACE_EN_2 constraints identified in Section 2.4.1.

Example (CDRefinement): the attribute disjointness constraint is bound to be violated when updating a class without updating its set of attributes (as a result two classes with the same set of attributes will appear in the transformation instance). To prevent this violation and still ensure that attributes of classes are disjoint in the source instance and target instance, the disjointness constraint in the source module is replaced by the following predicate:

```

pred attrDisj(context:set univ){
  no disj x,y: ((Class+AssociationClass) & context) | x.(attrs& context->context) & y.(
    attrs& context->context) !=none
}

```

In this predicate, all the sets and tuples of atoms present in the formula are restricted to the context given as parameter.

This predicate is then called in the transformation module with parameters consisting of all the atoms present in the source and target instance, respectively. Those set of atoms can be defined in Alloy, following the definition of f_{in} and f_{out} given in definition 7 and 8. Following is how those sets of atoms are defined for our CDRefinement case study:

```

let input = univ - (CREATE + DELETE + UPDATE + UPDATE.fixAbstract[Class] + CREATE.
  associationClass2Class[AssociationClass] + CREATE.newAssociations[Class, Association
  ])
let output = univ - (CREATE + DELETE + UPDATE + DELETE.(associationWithClass +
  associationClass) + UPDATE.fixAbstract.Class)

```

Those modifications entailed by Constraints Framing can be generated following the present procedure:

```

LET in = " let input= univ - ( CREATE + DELETE + UPDATE "
LET out = " let output= univ - ( CREATE + DELETE + UPDATE "
FOR EACH mapping map declared IN f DO
  WITH map: "map: X -> ... ->Xn -> Y"
  LET sig= signature in which map is declared
  IF sig = "CREATE" or sig = "UPDATE" THEN
    in+= "+ sig.map[X,...,Xn]"
  FI
  IF sig = "UPDATE" THEN
    out+= "+ sig.map.X(...).Xn"
  FI
  IF sig = "DELETE" THEN
    out+= "+ sig.map"
  FI
DONE
in += ")"
out += ")"
WRITE IN ConstraintFraming fact
  in
  out
WROTE
REPLACE facts by predicates in a_src
LET P be the set of added predicates
FOR EACH predicate p IN P DO
  LET n = p's name
  WRITE IN ConstraintFraming fact
    p[input] and p[output]
  WROTE
DONE

```

Note that replacing facts by predicates is done by creating, for each fact, a predicate taking an argument named `context` and of type set of `univ` (Alloy equivalent of `Object`) and by replacing each term t of the Alloy expression composing the fact by t & `context` $\rightarrow \dots \rightarrow$ `context`, the number of arrow separated `context` depending on the arity of expression t . We note that multiplicity constraints and other keywords such as “`disj`” should also be replaced by such predicates. The translation of those keywords into facts is given in [24].

- Constraints to be added to any f-module f defining an exogenous model transformation:

- **Minimum Output.** For an f-module $f : a_{\text{src}} \rightarrow a_{\text{dst}}$, we enforce that the set of atoms typed by signatures declared in a_{dst} should be limited to the output atoms of `CREATE` mappings declared in f . The intent is for the output of the transformation to be composed of only those elements defined through mappings, hence filling the role of the `C_TRACE_EX` constraint identified in Section 2.4.1. We note that this constraint does not apply to endogenous transformations because any a_{src} element not present in the range of a mapping is considered as part of the transformation’s source instance.

Example (CD2RDBMS): RDBMS elements are limited to the output atoms of declared mappings.

```

RDBMSElem= class2table[Class] + primAttr2column[Attribute] + classAttr2column[Attribute,
  Attribute] + association2column[Association,Attribute]

```

Minimum Output constraints can be generated following the present procedure:

```

FOR EACH signature Y declared IN a_dst DO
  LET M = FIND ALL mapping IN f having Y as range

```

```

LET constraint= "Y = "
IF M is empty THEN
    constraint += "none"
FI
FOR EACH map IN M DO
    WITH map: "map: X -> .. -> Xn -> Y"
    LET sig= signature in which map is declared
    constraint+= "sig.map[X,..,Xn]"
    IF not last iteration of this loop THEN
        constraint += "+"
    FI
DONE
WRITE IN Minimum Output fact
    constraint
WROTE
DONE
    
```

For any given f-module f , $a_f = \mathcal{T}(f)$ is the Alloy module composed of signatures fields and predicates declared in f and including the aforementioned constraints. We will see in the next subsection that those added constraints enforce f-modules to denote functions, that is, any Alloy module $\mathcal{T}(f)$ is a functional Alloy Module.

3.4.3 From F-modules to Functional Alloy Modules

We are now interested in proving that the Alloy module a_f obtained from the translation of an f-module f is a functional Alloy module. This enables us to ensure that F-Alloy specifications indeed denote functions.

To achieve our goal, we first define the transformation functions f_{in} and f_{out} used to obtain from an a_f -instance the source and target of the transformation denoted by f (See Definition 4). Next we examine the influence of rules defined in value predicates on the a_f -instance. We then show by construction that for any a_f , there cannot be two distinct a_f -instances x_f and x'_f s.t. $f_{\text{in}}(x_f) \neq f_{\text{in}}(x'_f)$ which will allow us to finally conclude that any Alloy module a_f obtained from the translation of an f-module f is a functional Alloy module (with respect to the transformation functions f_{in} and f_{out} defined).

Exogenous transformation functions

In the present subsection we define these functions for the exogenous case. In the next subsection we then deal with the endogenous case.

In the exogenous case we call the operation those functions embody instance projection and define it as follows:

Definition 6 (Instance Projection). *The projection of an instance $x : (X, Y, a)$ on a module $a' : (S', F', \varphi')$, with a' being in the import hierarchy of a , is the a' -instance composed of the atoms and tuples present in x and typed by signatures and fields of a' , respectively. We denote projections using the evaluation symbol \Downarrow : $x \Downarrow a'$ reads “the projection of x on a' ”.*

Formally : $x \Downarrow a' = (X', Y', a')$ with $X' = \{a^s \in X | s \in S'\}$ and $Y' = \{y^f \in Y | f \in F'\}$.

Thus for exogenous transformations:

$$f_{\text{in}}(x) = x \Downarrow a_{\text{src}} \text{ and } f_{\text{out}}(x) = x \Downarrow a_{\text{dst}}$$

We illustrate this application of instance projection in Fig. 3.11.

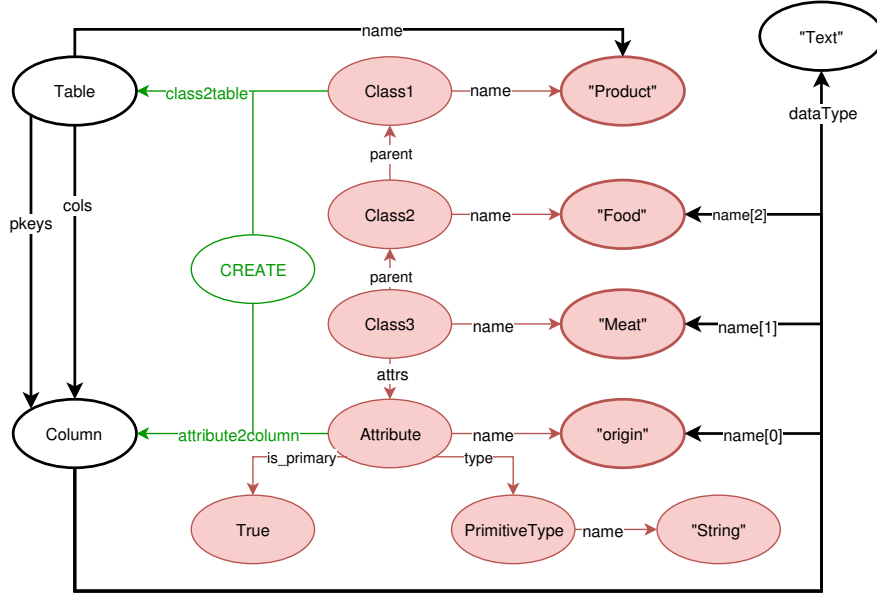


Figure 3.11: A CDRDBMS-instance x with $f_{\text{in}}(x)$ highlighted in red and f_{out} highlighted in bold

Endogenous transformation functions

We now define the functions f_{in} and f_{out} that extract the source instance and target instance from a a_f -instance in the endogenous case. We first give informal definitions and then provide formal definitions.

The function f_{in} extracts the source instance of the transformation from a given a_f -instance, namely, the subinstance made up of atoms typed by signatures in a_{src} and tuples typed by fields in a_{src} , with all atoms in the range of a CREATE or UPDATE mappings (and their associated tuples) being removed.

The function f_{out} extracts the target instance of the transformation from a given a_f -instance, namely, the subinstance made up of atoms typed by signatures in a_{src} and tuples typed by fields in a_{src} in which elements in the domain of UPDATE mappings are replaced by their images, preserving links to the replaced element, and where elements in the domain of DELETE mappings are removed.

We call those operations embodied by f_{in} and f_{out} “transformation-aware input projections” and “transformation-aware output projections”, respectively, and define them below.

in the following two definitions, we denote by m_{in} and m_{out} the set of input tuples and output atoms composing tuples typed by mapping m , respectively. We also write $a^s \in t^f$ if the atom a typed by signature s is contained in the tuple t typed by field f .

Definition 7 (transformation-aware input projection). *Given an instance $x_f : (X, Y, a_f)$, with $a_f : (S, F, \varphi)$ being a functional Alloy module expressing an endogenous transformation on $a_{\text{src}} : (S', F', \varphi')$, the f -aware input projection of x_f , denoted $x_f \Downarrow_{f_{\text{in}}}$ is the projection of x_f on a_{src} from which are subtracted output atoms of mappings in F .*

Formally : $x_f \Downarrow_{f_{\text{in}}} = x_f \Downarrow a_{\text{src}} - (X', Y', a_f)$ with :

- $X' = \{a^s \in X : \exists m \in F \text{ s.t. } a^s \in m_{\text{out}}\}$
- $Y' = \{t^f \in Y : \exists a^s \in X' \text{ s.t. } a^s \in t^f\}$

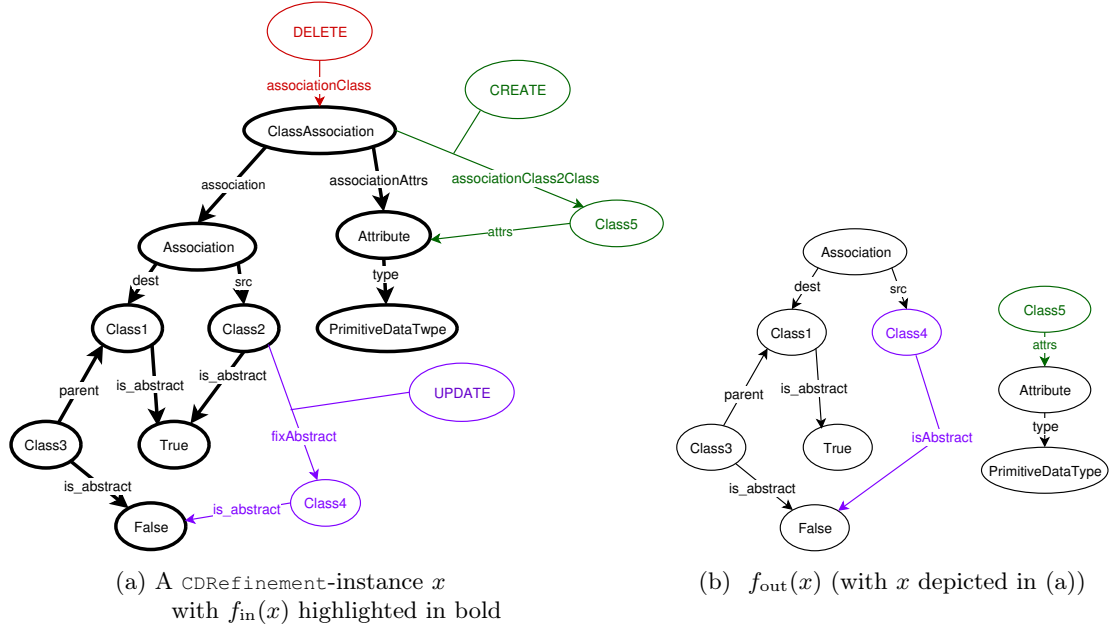


Figure 3.12: A CDRefinement-instance, and the input and output of the transformation CDRefinement specifies as defined by f_{in} and f_{out} , respectively.

In the following definition, we denote C , U and D the set of CREATE, UPDATE and DELETE mappings declared in a_f , respectively.

Definition 8 (transformation-aware output projection). *Given an instance $x_f : (X, Y, a_f)$ – with $a_f : (S, F, \varphi)$ expressing an endogenous transformation on $a_{src} : (S', F', \varphi')$ – the f -aware output projection of x_f , denoted $x_f \Downarrow_f^{out}$ is the projection of x_f on a_{src} from which are subtracted input tuples of UPDATE and DELETE mappings in F .*

Formally : $x_f \Downarrow_f^{out} = x \Downarrow a_{src} - (X', Y', a_{src}) + (\emptyset, Y'', a_{src})$ with :

- $X' = \{a^s \in X : \exists t^m \in Y \text{ s.t. } m \in U \cup D \wedge a^s \in m_{in}\}$
- $Y' = \{t^f \in Y : \exists a^s \in X' \text{ s.t. } a^s \in t^f\}$
- $Y'' = \bigcup_{t^m : (x_1, \dots, x_n) \in Y : m \in U \wedge \exists a^s \in m_{in} : a^s = x_i} (x_1, \dots, f(x_i), \dots, x_n)$

We then define for endogenous transformations:

$$f_{in}(x) = x \Downarrow_m^{in} \text{ and } f_{out}(x) = x \Downarrow_m^{out}$$

We illustrate the application of transformation-aware projections in Fig. 3.12. In those visualizations, atoms are represented by nodes and tuples by links. Note that the tuples typed by the UMLElement's field name have been filtered out for readability's sake.

The meaning of rules

Considering an f -module $f : a_{src} \rightarrow a_{dst}$, its associated Alloy module $a_f = \mathcal{T}(f)$ and $x_f \in \mathcal{Z}(a_f)$, the predicate association constraints defined previously enforce that, given a mapping μ declared in f , only those input tuples for which the guard predicate associated to μ is satisfied are associated through μ to an output atom. Moreover, the value predicate associated to μ holds in x_f when given as parameter any such pair of input tuples and output atom.

Each rule inside a value predicate represents a boolean expression that may use the parameters of the value predicate. In the following lemma we prove that each rule can be rewritten in the form

$$V \text{ in } f$$

where V is an Alloy expression denoting a relation, in denotes set inclusion, and f is a field of a_{dst} . Since V , seen as a set of tuples, generally depends on the rule r , the source instance $f_{\text{in}}(x_f)$, and the parameters x_1, \dots, x_n and y of the enclosing value predicate, we refer to V as $V(r, f_{\text{in}}(x_f), x_1, \dots, x_n, y)$.

To understand the importance of this lemma, note that the value predicate is a conjunction of rules, each stating that a set of tuples is contained in the relation representing a field of a_f . This will imply that the relation of each field can be written as the union of the V -sets of the contributing rules, since no other tuples can be in the relation of the field due to the minimal assignment constraints. This fact will be used to derive an explicit formula for the “shape” of a valid instance of a_f in Lemma 2.

Lemma 1. *Each rule r can be rewritten as a logically equivalent formula*

$$V(r, f_{\text{in}}(x_f), x_1, \dots, x_n, y) \text{ in } f$$

where V is an Alloy expression denoting a relation, in denotes set inclusion, and f is a field of a_{dst} .

Proof. We describe how to rewrite each type of rule in the required form. In the following, occurrences of name in each syntactic construct are replaced by y when they denote the output atom (according to the well-formedness of F-Alloy), or by f if they denote the field the rule is contributing to. The rewriting of rules is done as follows:

- For `strict` rules:

1. `y.f[[expr1]] = expr2`
2. `[expr1->]expr2 in y.f`
3. `y-> [expr1->]expr2 in f`

- For `loose` rules:

1. `y in image.f[[expr]]`
2. `[expr->]y in image.f`
3. `image-> [expr->]y in f`

- For `step` rules:

1. `all i:Int|range implies y.f[add[i,1]] = expr`
2. `all i:Int|range implies [add[i,1]]-> expr in y.f`
3. `all i:Int|range implies y-> [add[i,1]]-> expr in f`

□

We list in table 3.1 the set of tuples returned by the function V . For strict rules `expr1` and `expr2` may use input parameters as well as features of a_{src} (by the well-formedness constraints `ExprWF` given in Section 3.3.4). Thus the V -set for strict rules only depends on the source instance and the parameter values. For loose rules, `image`

Type of Rule r	Syntax followed by r	value of $V(r, x_f, (x_1, \dots, x_n), y)$
strict	$y.f[[\text{expr}_1]] = \text{expr}_2$	$\{y, \text{expr}_1, \text{expr}_2\}^f$
loose	$y \text{ in image.f}[[\text{expr}]]$	$\{(\text{image}, \text{expr}, y)\}^f$
step	all $i:\text{Int} \text{range}$ implies $y.f[\text{add}[i, 1]] = \text{expr}$	$\bigcup_{\{i \in \text{range}\}} \{(y, i + 1, \text{expr})\}^f$
conditional	expr implies rule	$\begin{cases} V(\text{rule}, x_f, (x_1, \dots, x_n), y), & \text{if } x_f \models \text{expr} \\ \emptyset & \text{otherwise} \end{cases}$

 Table 3.1: Valuation of V for the different rule constructs identified in Listing 3.6

denotes the output atom for an input tuple given by the specified expressions, which depends only on the source instance and values of input parameters. Therefore, the tuples in the V -set for loose rules depend only on the source instance and the parameter values as well. The same holds for step rules since the expr satisfy similar restrictions and the range is given by an expression that depends on the source instance and values of input parameters. Since the conditional expression in conditional rules satisfy the same restrictions, the corresponding V -set only depends on the source instance and the values of the parameters.

We now exemplify the valuation of τ given in table 3.1 by explicitly listing the tuples assigned through rules of the value predicate associated to the mapping `attribute2column` (taken from the CD2RDBMS case study). To do so, let us consider the CD2RDBMS-instance x_f depicted in Fig. 3.11. Given the two parameters $a = \text{Attribute}\$0$ and $c = \text{Column}\$0$ and $x_{\text{src}} = f_{\text{in}}(x_f)$ we have:

- With r being the strict rule

```
c.dataType=(a.type.name="String" implies "TEXT" else "NUMBER")
```

$$\begin{aligned} V(r, x_{\text{src}}, a, c) &= \{c, a.type.name="String" \text{ **implies** } \\ &\quad \text{"TEXT" **else** "NUMBER"}\}^{\text{dataType}} \\ &= \{\text{Column}\$0, \text{"TEXT"}\}^{\text{dataType}} \end{aligned}$$

- With r being the strict rule

```
c.name[0]= a.name
```

$$\begin{aligned} V(r, x_{\text{src}}, a, c) &= \{c, 0, a.name\}^{\text{name}} \\ &= \{\text{Column}\$0, 0, \text{"origin"}\}^{\text{name}} \end{aligned}$$

- With r being the strict rule

```
c.name[1]=((a.~attrs.parent)!=none implies a.~attrs.name else none)
```

$$\begin{aligned} V(r, x_{\text{src}}, a, c) &= \{c, 1, ((a.~attrs.parent)!=none \text{ **implies** } \\ &\quad a.~attrs.name \text{ **else** none)}\}^{\text{name}} \\ &= \{c, 1, (\text{Class}\$1!=none \text{ **implies** } \\ &\quad \text{Class}\$2.name \text{ **else** none)}\}^{\text{name}} \\ &= \{\text{Column}\$0, 1, \text{"Meat"}\}^{\text{name}} \end{aligned}$$

- With r being the step rule

```
all i:Int | i>=1 and i< #(a.~attrs.^parent) implies c.name[add[i,1]]= c.name[i].~name.parent.name
```

$$\begin{aligned}
 V(r, x_{\text{src}}, a, c) &= \bigcup_{\{i \in [1, \#(a.\sim\text{attrs}.\hat{\text{parent}}) \setminus \{ \}] \}} \{(c, i + 1, \\
 &\quad c.\text{name}[i].\sim\text{name}.\text{parent}.\text{name})^{\text{name}}\} \\
 &= \bigcup_{i \in [1, 2[} \{(c, i + 1, \\
 &\quad c.\text{name}[i].\sim\text{name}.\text{parent}.\text{name})^{\text{name}}\} \\
 &= \{(c, 2, \text{"Meat"}.\sim\text{name}.\text{parent}.\text{name})^{\text{name}}\} \\
 &= \{(c, 2, \text{Class}\$2.\text{parent}.\text{name})^{\text{name}}\} \\
 &= \{(\text{Column}\$0, 2, \text{"Food"})^{\text{name}}\}
 \end{aligned}$$

- With r being the conditional rule

```
a.is_primary=True implies c in CREATE.class2table[a.~attrs.*parent].pkeys
```

$$\begin{aligned}
 V(r, x_{\text{src}}, a, c) &= \begin{cases} V(r_2, x_{\text{src}}, a, c) & \text{if } x_{\text{src}} \models a.\text{is_primary}=\text{True} \\ \emptyset & \text{otherwise} \end{cases} \\
 &= V(r_2, x_{\text{src}}, a, c)
 \end{aligned}$$

with r_2 being the loose rule:

```
c in CREATE.class2table[a.~attrs.*parent].pkeys
```

$$\begin{aligned}
 V(r, x_{\text{src}}, a, c) &= \{(\text{class2table}[a.\sim\text{attrs}.*\text{parent}], c)^{\text{pkeys}}\} \\
 &= \{(\text{Table}\$0, \text{Column}\$0)^{\text{pkeys}}\}
 \end{aligned}$$

- With r being the loose rule:

```
c in CREATE.class2table[a.~attrs.*parent].cols
```

$$\begin{aligned}
 V(r, x_{\text{src}}, a, c) &= \{(\text{class2table}[a.\sim\text{attrs}.*\text{parent}], c)^{\text{cols}}\} \\
 &= \{(\text{Table}\$0, \text{Column}\$0)^{\text{cols}}\}
 \end{aligned}$$

Module a_f is a functional Alloy module

Let us consider an f-module $f : a_{\text{src}} \rightarrow a_{\text{dst}}$ specifying an exogenous or endogenous transformation. The following lemma proves that valid instances of a_f have a particular “shape”, namely, they can be written as the union of the source instance, atoms and tuples representing the mappings, as well as the union of V -sets of contributing rules. This lemma constitutes the final stepping stone that will allow us to prove (in Theorem 1) that a_f is a functional Alloy module.

In the following, we introduce the notation \vec{x} as a shorthand for tuples of the form (x_1, \dots, x_n)

Lemma 2. *Any valid instance $x_f : (X, Y, a_f)$ satisfies the equation:*

$$x_f = f_{in}(x_f) \cup A \cup \bigcup_{b \in A, \mu \in M_b} F(\mu)$$

where

$$F(\mu) = \bigcup_{(b, \vec{x}, y)^\mu \in Y} \left(\{y\} \cup \{(b, \vec{x}, y)\} \cup \bigcup_{r \in \mu} V(r, f_{in}(x_f), \vec{x}, y) \right)$$

with

- A being the set of singleton atoms typed after the signatures declared in f (*CREATE*, and possibly *UPDATE* and *DELETE*),
- M_b being the set of mappings declared in the signature typing b ,
- $(b, \vec{x}, y)^\mu \in x_f$ denoting that tuple (b, \vec{x}, y) ranges over all tuples in x_f typed by the mapping μ with first component equal to b ,
- $r \in \mu$ denoting that rule r is declared in the value predicate associated to μ ,
- $V(r, f_{in}(x_f), \vec{x}, y)$ as defined in the previous subsection.

Note that tuples typed by delete mapping are of the form (b, \vec{x}) . Moreover no value predicate is associated to the mapping. This is why we can simply replace V_r by \emptyset in the given equation. Hence we would have for any μ declared in a value *DELETE* signature:

$$F(\mu) = \bigcup_{(b, \vec{x})^\mu \in x_f} \{(b, \vec{x})\}$$

Proof:

- $f_{in}(x_f)$ yields a subinstance of x_f by construction, in both endogenous and exogenous cases.
- The presence of elements of A in x_f is due to a syntactic constraint of F-Alloy. Indeed each signature declaration is preceded by the keyword *one* ensuring the presence of exactly one atom typed by it.
- The rest of the formula is enforced by predicate association constraints stating that for each signature in a_f , for each mapping, for each input tuple, an output atom y should be part of the mapping (hence the addition of tuple (b, \vec{x}, y)). This output atom y also enters in the composition of tuples conforming to fields declared in the signature typing y . Those tuples are returned by the V -set of each rule r declared in the value predicate associated to the mapping. Note that Map Injectiveness, Map Disjunction and IODisjunction constraints enforce every y to be disjoint from $f_{in}(x_f)$ and from each other.
- Minimum Output constraints prevent x_f from being composed of any other elements in the case of an exogenous transformation. In the case of an endogenous transformation, any other element which is not part of a mapping is in $f_{in}(x_f)$.

We have shown by construction that the equation given in Lemma 2 is bound to hold in any instance of an Alloy module a_f obtained by translation of an f-module. \square

Theorem 1 (F-modules translate to functional Alloy modules). *For any f-module f , the translated module a_f is a functional Alloy module with respect to the transformation functions f_{in} and f_{out} defined in Section 3.4.3 for both exogenous and endogenous transformations.*

Proof. Let x_f and x'_f be two instances of a_f with $f_{in}(x_f) = f_{in}(x'_f)$.

Let us take a look at the terms on the right-hand side of the top equation in Lemma 2.

The first term is the same for x_f and x'_f by the above assumption.

The second term A is the same for x_f and x'_f .

For the third term select a mapping μ . Note that the set of argument tuples \vec{x} that satisfy the guard of μ is the same in both cases because $f_{in}(x_f) = f_{in}(x'_f)$. Now fix one such argument tuple \vec{x} . The tuple (b, \vec{x}, y) is the same in x_f and x'_f . Finally the set of tuples $V(r, f_{in}(x_f), \vec{x}, y)$ is the same, for any rule r in x_f and x'_f .

We conclude that x_f and x'_f are identical. It follows that $f_{out}(x_f) = f_{out}(x'_f)$ and hence a_f is a functional Alloy module. \square

3.5 Interpreting F-Alloy Specifications

We have seen in the previous section that any instance x_f of an Alloy module $a_f = \mathcal{T}(f)$ (with f being an f-module) can be computed from its subinstance $f_{in}(x_f)$ (see Lemma 2). In this section, we present the procedure for performing this computation, a process we call *interpretation*.

3.5.1 Definition

We start by introducing the notation used to denote interpretation:

Notation 3 (Interpretation). *We denote the set of instances obtained by interpretation of an f-module $f : a_{src} \rightarrow a_{dst}$, given an a_{src} -instance x_{src} , by $\mathcal{I}(f, x_{src})$.*

Bellow we give an F-Alloy interpretation procedure, showing how $\mathcal{I}(f, x_{src})$ can be computed, in Listing 3.7. The interpretation of rules is given as a subprocedure in Listing 6.2.5.

We note that in the following pseudo-code, the function `Eval(AlloyExpression e, Instance x)` represents the method `A4Solution.eval` provided in the Alloy API¹. This method evaluates an expression e in an `A4Solution` (object representation of an instance) x and returns a set of atoms, tuples or a boolean value depending on the type of expression given as parameter.

We now prove, by the two following lemmas, that such interpretation indeed conforms to the translational semantics of F-Alloy (defined in Section 3.4).

Lemma 3 (Interpretation Soundness). *Consider an f-module $f : a_{src} \rightarrow a_{dst}$, and let x_{src} be a valid instance of a_{src} . The instance returned by $\mathcal{I}(f, x_{src})$ is a valid instance of $\mathcal{T}(f)$. Formally: for all natural number s , there exists a natural number s' , such that,*

$$\forall x_{src} \in \mathcal{Z}(a_{src}, s), \mathcal{I}(f, x_{src}) \subseteq \mathcal{Z}(\mathcal{T}(f), s')$$

Proof. Line 56 of the interpretation pseudocode (Listing 3.7) ensures that the instance yielded by the interpreter is a valid instance of the Alloy module a_f yielded by $\mathcal{T}(f)$ (by checking that constraints φ_f of a_f hold in the returned instance x_f). The lemma thus trivially holds. \square

¹<http://alloy.mit.edu/alloy/documentation/alloy-api/index.html>

```

1  /*INPUT : an f-module  $f : a_{src} \rightarrow a_{dst}$  and an  $a_{src}$ -instance  $x_{src}$ 
2  *OUTPUT: a valid f-instance  $x_f : (A, T, a_f)$  s.t.  $f_{in}(x_f) = x_{src}$  or NONE */
3  FUNCTION Interpretation(F-module  $f$ , Instance  $x_{src}$ )
4  WITH  $x_{src} : (A_{src}, T_{src}, a_{src})$ 
5  LET  $c = \text{new Atom}(\text{CREATE})$ ,  $u = \text{null}$ ,  $d = \text{null}$ 
6  LET  $a_{src} = f.\text{getImportedModule}(1)$  //1st open statement
7  WITH  $a_{src} = (S_{src}, F_{src}, \varphi_{src})$ 
8  LET  $a_{dst} = f.\text{getImportedModule}(2)$  //2nd open statement
9  WITH  $a_{dst} = (S_{dst}, F_{dst}, \varphi_{dst})$ 
10 IF  $a_{dst} = \text{none}$  THEN //endogenous case
11      $u = \text{new Atom}(\text{UPDATE})$ 
12      $d = \text{new Atom}(\text{DELETE})$ 
13 FI
14 // We will return  $x_f : (A, T, a_f)$ 
15 LET  $A = c + u + d + A_{src}$ 
16 LET  $T = T_{src}$ 
17 FOR EACH mapping map IN  $f$  DO
18     LET sig = signature in which map is declared
19     IF sig = "CREATE" THEN
20         WITH map: "map:  $X_1 \rightarrow \dots \rightarrow X_n \rightarrow Y$ "
21         FOR EACH tuple  $(x_1^{x_1}, \dots, x_n^{x_n})$  WITH  $x_i^{x_i}$  IN  $A_{src}$  DO
22             IF Eval(guard_map( $x_1^{x_1}, \dots, x_n^{x_n}$ ),  $x_{src}$ ) THEN
23                  $y = \text{new Atom}(Y)$ 
24                  $A = A + y$ 
25                  $T = T + (c, x_1, \dots, x_n, y)^{\text{map}}$ 
26             FI
27         DONE
28     ELSE IF sig = "UPDATE" THEN
29         WITH map: "map:  $X \rightarrow Y$ "
30         FOR EACH atom  $a^X$  in  $x_{src}$  DO
31             IF Eval(guard_map( $a^X$ ),  $x_{src}$ ) THEN
32                  $y = \text{new Atom}(Y)$ 
33                  $A = A + y$ 
34                  $T = T + (u, a, y)^{\text{map}}$ 
35             FI
36         DONE
37     ELSE IF sig = "DELETE" THEN
38         WITH map: "map:  $X$ "
39         FOR EACH atom  $a^X$  in  $x_{src}$  DO
40             IF Eval(guard_map( $a^X$ ),  $x_{src}$ ) THEN
41                  $T = T + (d, a)^{\text{map}}$ 
42             FI
43         DONE
44     FI
45 DONE
46 FOR EACH mapping map IN  $f$  DO // CREATE or UPDATE mappings
47     FOR EACH tuple  $t^{\text{map}}$  IN  $T$  DO
48         WITH  $t : (b, x_1, \dots, x_n, y)$  //  $b$  is either a create or update atom
49         FOR EACH rule  $r$  IN value_map DO
50              $T = T + \text{ProcessRule}(r, x_f, (x_1, \dots, x_n), y)$ 
51         DONE
52     DONE
53 DONE
54 LET  $x_f = (A, T, a_f)$ 
55 LET  $a_f = (S_f, F_f, \varphi_f)$ 
56 IF Eval( $\varphi_f, x_f$ ) THEN
57     RETURN  $x_f$ 
58 ELSE
59     RETURN NONE
60 END FUNCTION
    
```

Listing 3.7: F-Alloy Mapping Interpretation pseudo code

3.5. INTERPRETING F-ALLOY SPECIFICATIONS

```
1  /*INPUT : r the rule to be processed
2            $x_f$  instance in which expressions are evaluated
3            $(x_1, \dots, x_n)$  and  $y$  parameters given to the value predicate containing the rule
4
5  *OUTPUT: a set of tuples typed by the field assigned in  $r$ */
6  FUNCTION ProcessRule(Rule r, Instance  $x_f$ , Tuple  $(x_1, \dots, x_n)$  , Atom  $y$ )
7      LET solution=none // tuples to return
8      IF r IS strict Rule THEN
9          WITH r: "y.f[[expr1]]=expr2"
10         LET val = Eval(expr1,  $x_f$ )
11         LET val2 = Eval(expr2,  $x_f$ )
12
13         FOR EACH v IN val DO
14             FOR EACH v2 IN val2 DO
15                 solution = solution +  $(y, v, v2)^f$ 
16             DONE
17         DONE
18     FI
19     IF r IS step THEN
20         WITH r: "all i:Int[range implies y.f[add[i,1]] = expr"
21         LET val = Eval(range,  $x_f$ )
22         LET t = none // set of tuples
23         FOR EACH v IN val DO
24             LET val2 = Eval(expr,  $x_f$ )
25             LET i = Eval(add[v,1])
26             FOR EACH v2 In val2 DO
27                 t = t +  $(y, i, v2)^f$ 
28             DONE
29         DONE
30         RETURN t
31     FI
32     IF r IS conditional THEN
33         WITH c:"expr implies r2"
34         IF Eval(expr,  $x_f$ ) = True THEN
35             RETURN ProcessRule(r2,  $x_f$ ,  $(x_1, \dots, x_n)$  ,  $y$ )
36         FI
37     FI
38     IF r IS loose THEN
39         WITH r: "y in image.f[expr]"
40         LET val1 = Eval(image,  $x_f$ )
41         LET val2 = Eval(expr,  $x_f$ )
42         FOR EACH v1 IN val1 DO
43             FOR EACH v2 IN val2 DO
44                 RETURN  $(v1, v2, y)^f$ 
45             DONE
46         DONE
47     FI
48 END FUNCTION
```

Listing 3.8: F-Alloy Rule Interpretation pseudo code

Lemma 4 (Interpretation Completeness). *Consider an $f : a_{src} \rightarrow a_{dst}$. Any instance x_f obtained by analysis of $\mathcal{T}(f)$ can be constructed by interpretation of f given $f_{in}(x_f)$.*

Formally: for all natural number s , we have,

$$\forall x_f \in \mathcal{Z}(\mathcal{T}(f), s), \mathcal{I}(f, f_{in}(x_f)) = \{x_f\}$$

Proof. The interpretation closely follows the structure of the equation given in Lemma 2, which holds in all valid instances of $\mathcal{T}(f)$. Furthermore the `ProcessRule` function closely follows the definition of the V -sets given in Lemma 1. By construction, interpretation will thus yield an $\mathcal{T}(f)$ -instance x_f in which the equation given in the aforementioned lemma holds. \square

3.5.2 Complexity Analysis

Let us analyze the time complexity of interpretation. Let n denote the number of atoms in x_{src} . There are two outer for-loops in the interpretation code. The first outer for-loop (1.17 –1.45) involves a polynomial number of evaluations (in terms of n) of Alloy expressions (guard predicates). If we assume that the evaluation of Alloy expressions can be done in polynomial time in n - which can be shown by structural induction - then the overall time for the first outer loop will be at most polynomial in n . The second outer for-loop entails a polynomial number of invocations of the `ProcessRule` function. We claim that each call to `ProcessRule` terminates in polynomial time. To see this, note that `ProcessRule` itself evaluates at most a polynomial number of Alloy expressions, hence yielding our claim. We conclude that interpretation takes at most time polynomial in n . We thus expect interpretation to be more efficient than analysis. This will be confirmed empirically in the next section.

3.6 F-Alloy Interpretation Performance

In this section, we provide a first step towards evaluating our approach by answering through empirical means the following questions:

1. What are the benefits of F-Alloy, performance-wise, compared to Alloy?
2. How does F-Alloy compare to other existing model transformation approaches in terms of performance?

We note that the following experiments were performed on a machine running an Intel i5 CPU (3.20Ghz) with 16GB of RAM, and that all the files necessary at the reproduction of those experiments can be found online¹.

3.6.1 F-Alloy vs Alloy

To answer the first question, we compare the time needed by the Alloy analyzer and the F-Alloy interpreter to compute the `CD2RDBMS` and `CDRefinement` transformations. The manipulations performed to compute the transformations are the following:

First, a sample of CD instances of various size is generated using the Alloy analyzer. Then for each CD instance x of this sample:

¹<https://goo.gl/fyvqRg>

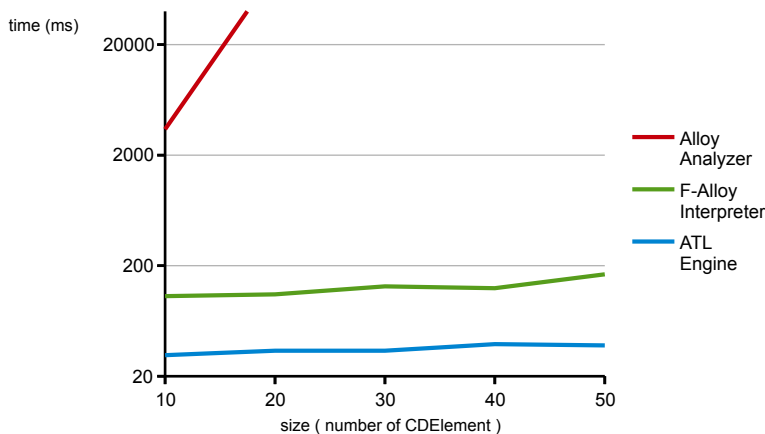


Figure 3.13: Time needed by the Alloy analyzer, the F-Alloy interpreter and the ATL engine to compute the `CD2RDBMS` transformation, given the same set of random CD-instances composed of 10 to 50 elements

In the case of Alloy analysis: we first add constraints to the `CD` module so that for any instance y of the analyzed transformation, $f_{in}(y) = x$. We then measure the time needed by the Alloy Analyzer (configured to use SAT4J) to obtain an instance of the `CD2RDBMS` and of the `CDRefinement` functional Alloy module, respectively. Functional Alloy modules are obtained by adding constraints to their f-modules counterparts following the procedures defined in Section 3.4.2.

In the case of F-Alloy interpretation: we measure the time needed by the F-Alloy interpreter to return an instance given as input the instance x and the `CD2RDBMS` and the `CDRefinement` f-module, respectively.

3.6.2 Comparing F-Alloy with Existing Model Transformations

To answer our second question, we compare the time needed by the F-Alloy interpreter to compute the `CD2RDBMS` and `CDRefinement` transformations to that of ATL (specification given in Annexe C) and Henshin (specification given in Section 3.2.3), respectively. The sample of source instances used is the same than the one on which the manipulations of the previous experiment were carried out.

3.6.3 Results

The time needed by the Alloy analyzer, the F-Alloy interpreter and commonly used transformation engines (ATL engine and Henshin interpreter) to compute the `CD2RDBMS` and `CDRefinement` transformation is plotted in Figures 3.13 and 3.14, respectively. The measurements were performed on CD-instances containing up to 50 atoms.

Observations: From those measurements, we clearly see that the F-Alloy interpreter drastically outperforms the Alloy analyzer in the exercise of computing both endogenous and exogenous model transformations. The F-Alloy interpreter’s performance is actually of the same order than those of the ATL engine and the Henshin interpreter hence providing empirical evidence that the F-Alloy interpreter can be used in practice to compute model transformations. We observe though that compared to Henshin and ATL which are mature model transformation languages, the time needed by F-Alloy to complete its tasks is less “stable” and varies more in function of the source instance’s

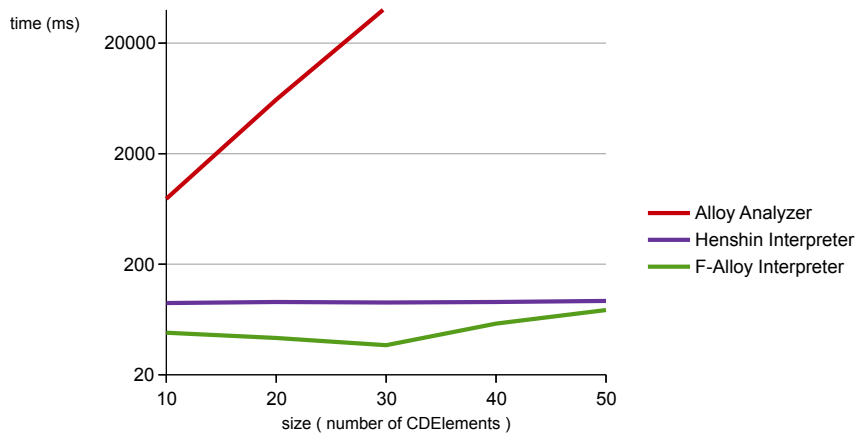


Figure 3.14: Time needed by the Alloy analyzer, the F-Alloy interpreter and the Henshin interpreter to compute the `CDRefinement` transformation, given the same set of random CD-instances composed of 10 to 50 elements

size. This can be explained by the difference of maturity between the tools supporting those languages and may also be partly attributed to the current lack of effort spent in optimizing the F-Alloy interpreter’s code.

3.7 Related Work on Model Transformation Languages

We can consider the F-Alloy language as a relational model transformation language. Relational model transformation languages (such as those given in [83], [75] and [84]) are those where the main concept is that of a mathematical relation [85]. Note that in F-Alloy the mathematical relations, represented by mappings, are in fact injective functions. In their pure form (e.g., [83]) relational specifications are not executable. In other cases (e.g., [75]) they are executable in principle but still lack proper tool support. In the case of QVTr there are some tools that execute QVTr specifications but none of them take into account all the features of the QVTr language yet. Note however that the QVTd project is working on a soon-to-be-finished implementation of QVTr¹. This is an indication that providing execution semantics for a relational language is a non-trivial task, especially if some semantic inconsistencies exist as is the case for QVTr ([43]). In this chapter we have shown that F-Alloy specifications are efficiently executable.

In the case of endogenous transformations, in-place transformations – defining how to obtain the target model of the transformation via operations to be applied on a given source model – are often considered as syntactic sugar for their out-place counterparts – defining how to build the target model from scratch given a source model. To illustrate this trend, the ATL refining mode, which allows the specification of in-place transformation, is executed as an out-place transformation (source model not modified, elements copied from source to target model) by the ATL 2004 compiler. Our motivation to make in-place constructs an integral part of our functional Alloy modules, and consequently of F-Alloy is to directly provide a formal ground to F-Alloy’s in-place syntax, thus preventing any ambiguities in the semantics of each operation. Another language that natively supports endogenous in-place transformations is Henshin [82], which was

¹<https://projects.eclipse.org/projects/modeling.mmt.qvtd>

used in the case study section as well as in the evaluation section and supports a formal graph transformation semantics.

One distinguishing feature of F-Alloy is that it inherits a formal semantics from the host language Alloy (although one could argue that the use of pseudo-code in the translation procedure yields a semi-formal description of the semantics of F-Alloy). Not all model transformation languages are formal. For instance the previously mentioned model transformation language ATL [86] was defined semi-formally. A formal semantics in terms of rewriting logics was later given by [87]. Even if a formal semantics is given there is in general no guarantee that the implementation does indeed conform to the semantics. A good illustration of this is the case of the triple graph grammar approach [74, 88], used in Section 3.2, for which the authors of [89] describe an approach to show conformance of an existing implementation to the formal semantics.

3.8 Summary

In this chapter we have introduced a new language named F-Alloy enabling the concise expression of functional Alloy modules and allowing the efficient computation of the transformation they define through a process we called *interpretation*.

We have given evidence of F-Alloy benefits, namely conciseness and efficient computation, by experimenting on two case studies, the previously defined `CD2RDBMS` and `CDRefinement` transformations. We have defined the semantics of F-Alloy through the specification of a translation into Alloy and have shown that the F-Alloy interpretation conforms to this translation in the sense that interpretation of an f-module (given a source instance) yields the same instance than the analysis of its semantically equivalent functional Alloy module (fixing the source instance).

F-Alloy inherits the formal semantics of Alloy, thus making the specified model transformations analyzable. This contrasts with other approaches where a separate formal semantics has to be defined. It is worth mentioning that Alloy based analysis, though successfully applied to a plethora of domains – from software architecture [90, 91] to information security [92, 93] –, still lack the ability of producing intuitive visual feedbacks, necessary to enhance review-based validation experiences. In the next chapter, we propose a solution to enhance the intuitiveness of those visual feedbacks used in the context of model transformation specifications debugging.

Chapter 4

On Model Transformation Validation

In the previous chapter, we have introduced the F-Alloy language as an Alloy-based DSL aiming at concisely specifying efficiently interpretable model transformations. In this Chapter, we introduce *hybrid analysis*, an approach combining F-Alloy interpretation and Alloy analysis so as to speed up the analysis of F-Alloy specifications. We also propose a novel approach to the validation of model transformations, namely *Visualisation Based Validation* (VBV), relying on hybrid analysis and allowing domain experts to be part of the transformation validation process. This approach requires the introduction of a new kind of transformation, namely compound transformations, also introduced in this chapter. The extension of F-Alloy to support the specification of compound transformations is defined in Section 4.2. We then coin the notion of hybrid analysis in section 4.3 and introduce Visualization Based Validation in Section 4.4. An illustration of this novel validation approach is provided in section 4.5, and an evaluation of hybrid analysis performance is given in Section 4.6. Finally, we conclude this chapter by discussing related work on model transformation validation in Section 4.7 and by providing a summary of this chapter’s contribution in Section 4.8.

4.1 Validation of F-Alloy Specifications

In the previous chapter, we have introduced F-Alloy, an Alloy-based DSL designed to enable the concise expression of functional Alloy modules. We have also introduced the F-Alloy interpreter, which enables, given a source instance the efficient computation of a target instance. The semantics of F-Alloy being defined through a translation to Alloy, Alloy-based validation techniques (relying on the Alloy analyzer) can be seamlessly applied to F-Alloy specifications. Indeed, to validate an f-module, the naive approach would be add constraints to that f-module to obtain the corresponding functional Alloy module on which Alloy analysis can be performed. This approach suffers two limitations.

- The domain space in which analysis of functional Alloy modules is performed is inherently big hence preventing analysis completion in reasonable time. This limitation has already been mentioned (see **L.1** Section 2.4.3) when reasoning about the fitness of Alloy analysis in computing model transformations.
- The visual feedback returned by Alloy analysis can hardly be used to validate the transformation as its readability is (1) hindered by the amount of elements present in the visualization and (2) independent of the domain the transformation is used in.

In this chapter, we address those two limitations as follows:

- To improve the efficiency of F-Alloy specification analysis, we define an approach, named *hybrid analysis* combining Alloy analysis with F-Alloy interpretation.
- To improve the effectiveness of F-Alloy specification analysis, we propose a new process, called Visualization Based Validation (VBV), which relies on defining a visualization for those instances returned by hybrid analysis. This allows (1) a more concise and more intuitive representation of transformation executions and (2) the definition of domain specific visualization so as to involve domain experts in the model transformation validation process.

The proposed VBV approach relies on an F-Alloy extension which enables the specification of *compound transformations*, that is, transformations that take a transformation specification (in our case an f-module) as input. We note that compound transformations are not to be mistaken with higher-order transformations as introduced in [94], which are transformations taking a metamodel of a transformation language (e.g., the ATL metamodel) as source or target metamodel. We do not study higher-order transformations in this work as the metamodels defining a transformation language can be expressed as an Alloy module, making their expression natively supported by F-Alloy.

We note that solely exogenous transformations are used to illustrate this chapter for conciseness sake. Nevertheless all contributions, namely compound transformations, hybrid analysis and VBV, can also be applied to endogenous transformations.

4.2 Extending F-Alloy to Specify Compound Transformations

F-Alloy, as defined in previous chapter, only supports the specification of transformations having as source and target modules Alloy modules. In this section, we extend F-Alloy to allow the specification of compound transformations, having f-modules as source and/or target modules. This extension will, in addition to enabling the reuse of existing model transformation specifications, constitute the cornerstone of a new approach to model transformation validation (see Sect. 4.4).

4.2.1 Syntactic Extension

Fig. 4.1, from left hand side to right hand side, conceptually summarizes the consequences of the syntactic change brought to F-Alloy. This syntactic change is realized by modifying the **ImportWF** well-formedness constraint from the F-Alloy definition given

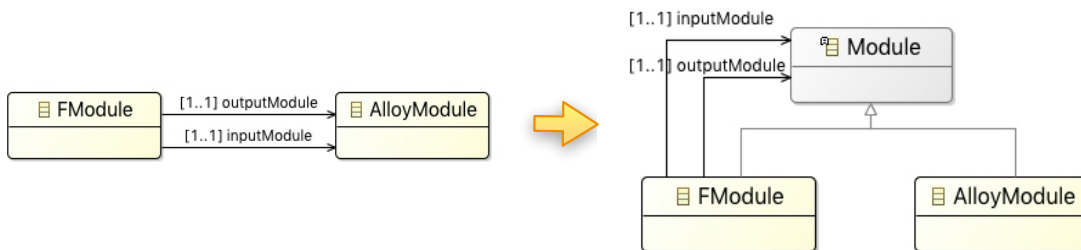


Figure 4.1: Extending F-Alloy (from left to right) to support both basic and compound f-modules

in Section 3.3.4. We give in the following the refined version of the **ImportWF** well-formedness constraint highlighting, using ~~strikeout~~ and **boldfaced** fonts, parts that are removed from and added to the constraint, respectively:

ImportWF (reworked) In the `fmodule` rule, there is exactly one `import` occurrence in the case the f-module defines an endogenous transformation and exactly two `import` occurrences in the case the f-module defines an exogenous transformation. In these `import` rules, `qualName` should refer to **a module being either expressed in Alloy or in F-Alloy** ~~an Alloy module~~. In the case of an exogenous transformation, the two referred ~~Alloy~~ modules should be distinct. Those modules are named *source module* and *target module*, respectively (as they represent the source and target metamodels of the specified transformation).

F-modules and Alloy modules are now unified under a common supertype referred to as *modules*. In the following, we keep using a and f to range over Alloy modules and f-modules, respectively, and use m to range over both of them m . An f-module $f : m_{\text{src}} \rightarrow m_{\text{dst}}$ can thus either be a basic or compound f-module.

We now distinguish two kinds of f-modules:

Basic f-modules are f-modules specifying a model transformation whose source and target modules are both Alloy modules.

Compound f-modules are f-modules specifying a compound model transformation, i.e., whose source and/or target modules are themselves f-modules.

Updating the ImportWF constraints drives us to update the translational semantics of F-Alloy so as to preserve the intended meaning of the imported f-module.

4.2.2 Semantic Extension

In the previous chapter, we have defined a translational semantics to F-Alloy, defining how to transform a basic f-module $f : a_{\text{src}} \rightarrow a_{\text{dst}}$ to a plain Alloy module a_f . We recall that this translation is denoted by $\mathcal{T}(\bullet)$ and that it amounts to the systematical addition of constraints following the templates identified in Section 3.4.2.

This translational semantics needs to be adapted to successfully define the intended meaning of compound f-modules. To avoid ambiguity, from now on, we refer to the translational semantics previously defined for basic f-modules (denoted $\mathcal{T}(\bullet)$) as the *basic translation*.

Definition 9 (Extended Translational Semantics). *Let $[\bullet]_{\mathcal{T}}$ denote the extended semantics. Given a module m – m being an Alloy module, basic f-module or compound f-module – $[m]_{\mathcal{T}}$ denotes the Alloy module giving its meaning to m . Depending on whether m is an Alloy module, a basic f-module or a compound f-module, we define:*

$$[f : m_{\text{src}} \rightarrow m_{\text{dst}}]_{\mathcal{T}} \stackrel{\text{def}}{=} \mathcal{T}(f : [m_{\text{src}}]_{\mathcal{T}} \rightarrow [m_{\text{dst}}]_{\mathcal{T}})$$

$$[a]_{\mathcal{T}} \stackrel{\text{def}}{=} a$$

Briefly, in case m is a basic f-module, the basic translation ($\mathcal{T}(\bullet)$) is applied. In case m is a compound f-module, the imported modules need to be translated recursively (by applying $[\bullet]_{\mathcal{T}}$) prior to applying the basic translation. Based on this extended translational semantics, we define instances of a module m as follows:

Definition 10 (Module Instances). *Given a module m , an instance x is a (valid) instance of m if and only if x is a (valid) instance of $[m]_{\mathcal{T}}$. We then call x an m -instance.*

We recall the existence of transformation functions f_{in} and f_{out} yielding the source and target instance of a transformation defined by a module m , given an m -instance. Those transformations are defined for exogenous and endogenous transformations in Section 3.4.3 and 3.4.3, respectively. Let us keep in mind for the remainder of this section that the function f_{in} has the following property:

Lemma 5 (f_{in} Property). *Given an f -module $f : m_{\text{src}} \rightarrow m_{\text{dst}}$, a scope s and an f -instance $x \in \mathcal{Z}([f]_{\mathcal{T}}, s)$, the following property holds:*

$$f_{\text{in}}(x) \in \mathcal{Z}([m_{\text{src}}]_{\mathcal{T}}, s)$$

Proof. The computation of $f_{\text{in}}(x)$ consists, in both the case of exogenous (defined in Section 3.4.3) and endogenous (defined in Section 3.4.3) transformations, in removing elements from x . Thus, it holds by construction that any instance yielded by $f_{\text{in}}(x)$ is a sub-instance of x . \square

The F-Alloy interpretation procedure, already defined in Section 3.5 can be used as is¹ on compound f -module while still being sound and complete w.r.t. F-Alloy's refined translational semantics.

We provide a uniform definition of soundness and completeness for (basic and compound) f -modules in the following lemma.

Lemma 6 (Soundness and Completeness of Compound f -module interpretation). *The interpretation of an f -module $f : m_{\text{src}} \rightarrow m_{\text{dst}}$ is (1) sound and (2) complete. That is, for all natural number s , there exists a natural number s' , such that:*

(1)

$$\forall x_{\text{src}} \in \mathcal{Z}([m_{\text{src}}]_{\mathcal{T}}, s), \mathcal{I}(f, x_{\text{src}}) \subseteq \mathcal{Z}([f]_{\mathcal{T}}, s')$$

(2)

$$\forall x_f \in \mathcal{Z}([f]_{\mathcal{T}}, s), \mathcal{I}(f, f_{\text{in}}(x_f)) = \{x_f\}$$

Proof. Soundness has been proven for the interpretation of basic f -modules in Lemma 3. Concerning the soundness of compound f -module interpretation, occurrences of a_f in the interpretation pseudo code (Listing 3.7) denote the Alloy module giving its meaning to f . The translational semantics of F-Alloy being redefined, it is understood that in the case of f being a compound f -module, $a_f = [f]_{\mathcal{T}}$ and not $\mathcal{T}(f)$. The check performed on line 56 of the interpretation pseudo code thus also effectively ensures the soundness of compound f -module interpretation.

Completeness has been proven for the interpretation of basic f -modules in Lemma 4 by showing that the interpretation follows the constraints defined in the basic semantics. Interpretation following the exact same process for compound model transformation, it still follows those constraints. Completeness thus also holds for compound f -modules. \square

Interpretation working for both basic and compound f -modules, we use the same notation (Notation 3) to denote the interpretation of basic f -modules and compound f -modules.

¹Considering the pseudocode given in Listing 3.7, the only adaptation required by the interpretation of compound f -module is to replace $f : a_{\text{src}} \rightarrow a_{\text{dst}}$ by $f : m_{\text{src}} \rightarrow m_{\text{dst}}$ so that the imported modules can now also be f -modules)

4.3 Hybrid Analysis of (Compound) Model Transformations

In the previous section, we have introduced the extended semantics of F-Alloy through the definition of the translation function $[\bullet]_{\mathcal{T}}$. With this translation yielding an Alloy module, it is possible to generate the set of instances of an f-module for validation purposes. The straightforward, but time consuming, way of obtaining this set of instances consists in first translating the f-module to the corresponding augmented module, then performing Alloy analysis on the latter.

In this section, we propose a more efficient approach to the problem of instance generation. Instead of relying on pure SAT-based analysis like the Alloy analyzer, we propose a hybrid strategy that combines both SAT-based analysis and F-Alloy interpretation.

4.3.1 Definition

We recall that the goal of this section is to define an efficient way to obtain instances conforming to f-modules for validation purposes. We achieve this goal by proposing a hybrid strategy combining Alloy analysis with the previously introduced F-Alloy interpretation. We call this new hybrid approach *hybrid analysis* and define it as follows:

Definition 11 (Hybrid Analysis). *We denote the set of instances obtained by hybrid analysis of a module m within a scope s by the mathematical function $\mathcal{H}(m, s)$, defined as follows:*

$$\mathcal{H}(m, s) = \begin{cases} \mathcal{Z}(m, s) & \text{if } m \text{ is an Alloy module} \\ \bigcup_{x_{\text{src}} \in \mathcal{H}(m_{\text{src}}, s)} \mathcal{I}(m, x_{\text{src}}) & \text{if } m \text{ is an f-module} \end{cases}$$

Briefly, given a module m , if m is an f-module from m_{src} to m_{dst} , the set of m -instances is obtained by first applying the hybrid analysis on m_{src} , then applying the F-Alloy interpretation on m for each instance of m_{src} obtained previously. Otherwise, if m is a plain Alloy module, the set of m -instances is obtained by applying Alloy's analysis to m .

We demonstrate the correctness of the hybrid analysis strategy by the following theorem, which basically states that the result obtained by applying the hybrid analysis is equivalent to the result obtained by applying Alloy's analysis.

Theorem 2 (Correctness of Hybrid Analysis). *Given a module m and its corresponding augmented module $[m]_{\mathcal{T}}$, for every natural number s , there exists another natural number s' , such that $\mathcal{H}(m, s) = \mathcal{Z}([m]_{\mathcal{T}}, s')$.*

Proof. By Definition 11 the hybrid analysis of an f-module is a recursive process, i.e., in case of an hybrid analysis of a compound f-module from m_{src} to m_{dst} a hybrid analysis of m_{src} is performed first. Knowing that the import hierarchy is acyclic (property inherited from Alloy), we prove the correctness of the above theorem by structural induction.

- If m is a plain Alloy module, then:
 1. $\mathcal{H}(m, s) = \mathcal{Z}(m, s)$ (Definition 11)
 2. $m = [m]_{\mathcal{T}}$ (Definition 9)
 3. $\mathcal{H}(m, s) = \mathcal{Z}([m]_{\mathcal{T}}, s)$ (1. and 2.)
 4. the theorem holds (3.)
- If m is a basic f-module from a_{src} to a_{dst} , then:

1. $\mathcal{H}(m, s) = \bigcup_{x_{\text{src}} \in \mathcal{H}(a_{\text{src}}, s)} \mathcal{I}(m, x_{\text{src}})$ (Definition 11)
2. $\mathcal{H}(a_{\text{src}}, s) = \mathcal{Z}(a_{\text{src}}, s)$ (Definition 11)
3. $\mathcal{H}(m, s) = \bigcup_{x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s)} \mathcal{I}(m, x_{\text{src}})$ (1. and 2.)
4. $\forall x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s), \mathcal{I}(m, x_{\text{src}}) \subseteq \mathcal{Z}([m]_{\mathcal{T}}, s')$ (equation (1) of Lemma 6)
5. $\mathcal{H}(m, s) \subseteq \mathcal{Z}([m]_{\mathcal{T}}, s')$ (3. and 4.)
6. $\forall x \in \mathcal{Z}([m]_{\mathcal{T}}, s'), \{x\} = \mathcal{I}(m, f_{\text{in}}(x))$ (equation (2) of Lemma 6)
7. $\mathcal{Z}([m]_{\mathcal{T}}, s') \subseteq \left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{T}}, s')} \mathcal{I}(m, f_{\text{in}}(x)) \right)$ (6.)
8. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{T}}, s')} \{f_{\text{in}}(x)\} \right) \subseteq \mathcal{Z}(a_{\text{src}}, s')$ (Lemma 5)
9. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{T}}, s')} \{f_{\text{in}}(x)\} \right) \subseteq \left(\bigcup_{x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s')} \{x_{\text{src}}\} \right)$ (8.)
10. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{T}}, s')} \mathcal{I}(m, f_{\text{in}}(x)) \right) \subseteq \left(\bigcup_{x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s')} \mathcal{I}(m, x_{\text{src}}) \right)$ (9.)
11. $\mathcal{Z}([m]_{\mathcal{T}}, s') \subseteq \left(\bigcup_{x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s')} \mathcal{I}(m, x_{\text{src}}) \right)$ (7. and 10.)
12. $\mathcal{Z}([m]_{\mathcal{T}}, s') \subseteq \mathcal{H}(m, s')$ (3. and 11.)
13. $\mathcal{H}(m, s) = \mathcal{Z}([m]_{\mathcal{T}}, s')$ (5 and 12.)
14. the theorem holds

- If m is a compound f-module from m_{src} to m_{dst} , then:

1. $\mathcal{H}(m, s) = \bigcup_{x_{\text{src}} \in \mathcal{H}(m_{\text{src}}, s)} \mathcal{I}(m, x_{\text{src}})$ (Definition 11)
2. $\mathcal{H}(m_{\text{src}}, s) = \mathcal{Z}([m_{\text{src}}]_{\mathcal{T}}, s')$ (induction hypothesis)
3. $\mathcal{H}(m, s) = \bigcup_{x_{\text{src}} \in \mathcal{Z}([m_{\text{src}}]_{\mathcal{T}}, s')} \mathcal{I}(m, x_{\text{src}})$ (1. and 2.)
4. let $[m_{\text{src}}]_{\mathcal{T}} = a_{\text{src}}$ ($[m_{\text{src}}]_{\mathcal{T}}$ being an Alloy module).
5. $\mathcal{H}(m, s) = \bigcup_{x_{\text{src}} \in \mathcal{Z}(a_{\text{src}}, s')} \mathcal{I}(m, x_{\text{src}})$ (3. and 4.)
6. The equation in 5. is the same than the one given in point 3. of the reasoning for basic f-module. Hence, following the same reasoning as for basic f-modules, the theorem holds for compound f-modules.

We have proven, case by case, that the hybrid analysis of any module, independently of its nature, is equivalent to the Alloy analysis of its corresponding augmented module (in the sense that it produces, for given scopes, the same set of instances).

□

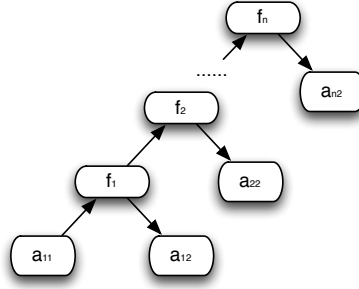


Figure 4.2: Import Hierarchy of an f-module

4.3.2 Complexity Analysis

Compared to pure Alloy based analysis, hybrid analysis substantially reduces computational complexity. Let f_n be an f-module with its transformation hierarchy depicted in Fig. 4.2. Instead of analyzing $[f_n]_{\mathcal{T}}$ using the Alloy Analyzer, which is very resource consuming especially in case of big scopes, one only needs to analyze the left-most leaf Alloy module a_{11} and all the rest can be built by applying the F-Alloy interpretation to f_1, \dots, f_n , each taking polynomial time (see Section 3.5.2). In addition, the problem of finding an optimal scope for a given analysis is also reduced accordingly: one only needs to define a scope for a_{11} instead of f_n which is much more complex. More specifically, the time complexity of hybrid analysis is given below.

Proposition 1 (Hybrid Analysis Complexity). *The time needed for hybrid analysis of an f-module f from m_{src} to m_{dst} is a polynomial function of the time needed for the hybrid analysis of m_{src} .*

Proof. Let $t(m_{src})$ denote the time needed for the hybrid analysis of m_{src} and let $t(f)$ denote the time needed for the hybrid analysis of f . From Definition 11 it follows that:

$$t(f) \leq c \cdot \sum_{x_{src} \in \mathcal{H}(m_{src}, s)} t_{\mathcal{I}}(m_{src}, x_{src})$$

where $t_{\mathcal{I}}(m_{src}, x_{src})$ is the time for the interpretation of f on input x_{src} and c is a constant. By the results of the complexity analysis performed in Section 3.5.2, and the fact that interpretation is independent of the nature of the imported modules, there exists a polynomial p such that $t_{\mathcal{I}}(m_{src}, x_{src}) \leq p(|x_{src}|)$ where $|x_{src}|$ is the size of instance x_{src} . The instance x_{src} being obtained from hybrid analysis of m_{src} , it follows that $|x_{src}| \leq t(m_{src})$. Therefore:

$$t(f) \leq c \cdot \sum_{x_{src} \in \mathcal{H}(m_{src}, s)} p(t(m_{src}))$$

Since $|\mathcal{H}(m_{src}, s)| \leq t(m_{src})$, we conclude that:

$$t(f) \leq c \cdot t(m_{src}) \cdot p(t(m_{src}))$$

thus implying the proposition. \square

4.4 Visualization-Based Validation of Model Transformation

In the previous section, we have introduced hybrid analysis as a way to efficiently obtain the set of all instances of a given f-module. The fact that hybrid analysis can be applied

to compound transformations of any depth paves a new way to the validation of F-Alloy transformations, namely Visualization-Based Validation (VBV), complementing the other existing model transformation validation techniques [95].

Given a transformation f specified in F-Alloy, the VBV of f relies on the definition of a compound transformation called f_viz from f to VLM as depicted in Fig. 4.3, to give a domain specific visualization to f -instances.

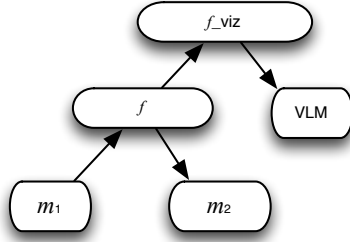


Figure 4.3: Import Hierarchy of f_viz

VLM, standing for Visual Language Metamodel, is an Alloy module (provided in Annexe D) defining a set of graphical concepts such as shapes (e.g., `RECTANGLE`, `ELLIPSE`), colors, layouts, and arrows/lines allowing to connect shapes (i.e., `CONNECTOR`). A tool, that we will present in the next chapter, can be used to parse a VLM-instance and render accordingly the graphical representation it denotes. We note that in the following, given a module m (specified in Alloy or F-Alloy), the notation m_viz denotes a transformation from m to VLM that provides a domain specific visualization to m -instances.

In Fig. 4.4, we provide an iterative process prescribing an effective application of VBV to a model transformation $f : m_{src} \rightarrow m_{dst}$. We note that two actors are part of this process:

- The *Domain Expert* is acquainted with what is modeled by the source and target metamodel of the transformation to be validated and knows which target model should the transformation yield for a given source model.
- The *Transformation Engineer* is an expert in Alloy and F-Alloy and implements the transformation f following his/her understanding of the transformation as conveyed by the domain expert.

The first step of this process is to define the f_viz transformation. It is composed of a set of mappings defining (1) the domain specific visualization of m_{src} and m_{dst} , and (2) the visualization of applications of mappings declared in f – e.g., *traces* relating m_{src} elements to m_{dst} elements.

For (1), the domain expert provides instructions on how m_{src} and m_{dst} instances should be represented to the transformation engineer. The transformation engineer then specifies the transformations m_{src_viz} (step 1.1) and m_{dst_viz} (step 1.2). Note that those transformations might already exist under certain circumstances (e.g., a transformation from (or to) m_{src} (or m_{dst}) has already been validated using VBV).

For (2), we propose a default visualization of traces: dashed arrows with distinct colors linking elements of m_{src} to the elements of m_{dst} . This default visualization can be automatically defined by the following rules :

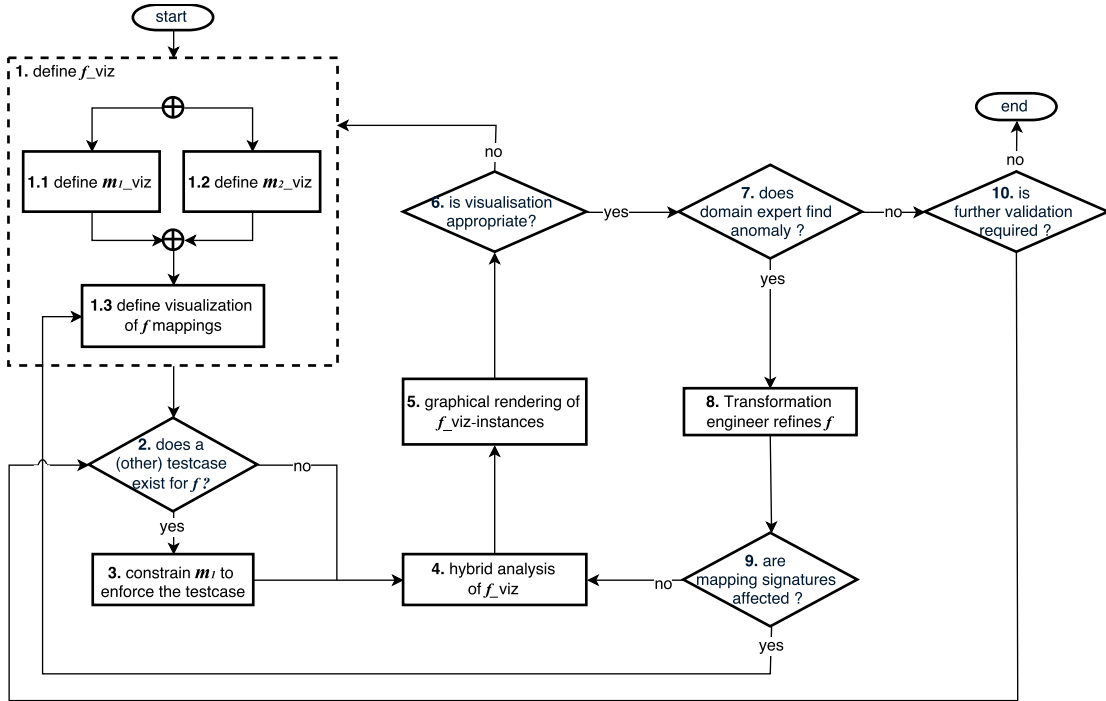


Figure 4.4: Visualization-Based Validation (VBV) of the model transformation $f : m_{\text{src}} \rightarrow m_{\text{dst}}$

- For each mapping $\mu_i : X \rightarrow Y$ of f , a mapping $v_i : X \rightarrow Y \rightarrow \text{CONNECTOR}$ is added to the f_viz transformation.
- The guard predicate associated to mapping v_i specifies that only pairs of elements of X and Y that are part of mapping μ_i yield a new **CONNECTOR**.

```

pred guard_vi(x:X, y:Y) {
    CREATE->x->y in  $\mu_i$ 
}
    
```

- The value predicate associated to mapping v_i specifies that each created **CONNECTOR** has as source the graphical representation of x as defined in $m_{\text{src_viz}}$ and as target the graphical representation of y as defined in $m_{\text{dst_viz}}$. Moreover the **CONNECTOR** is labeled and colored accordingly to indicate that it represents a trace of μ_i .

```

/* mapping XtoViz (resp. YtoViz) is defined in m_src_viz (resp. m_dst_viz)
and provide a graphical representation of x (resp. y) */
pred value_vi(x:X, y:Y, c:CONNECTOR) {
    c.source = CREATE.XtoViz[x]
    c.target = CREATE.YtoViz[y]
    c.color = RED
    c.label = " $\mu_i$ "
}
    
```

Once f_viz is defined, one has to decide how VBV will be performed (step 2). VBV can be performed either (1) on concrete test case, or (2) on random instances within a finite scope.

In case of (1), the transformation engineer adds constraints to m_{src} so that the only possible m_{src} -instances satisfying those constraints are the ones described in the test

cases (step 3). Those constraints can be systematically generated given the test cases to enforce (see an example given in Listing 4.3).

In case of (2), the transformation engineer provides a scope to elements of m_{src} (or the left-most leaf Alloy module in the import hierarchy of m_{src} in case m_{src} is also an f-module). Note that the effectiveness of this case is based on the small scope hypothesis [96] claiming that most of design errors can be reproduced in small instances.

Once the m_{src} module has been constrained as per step 3, a hybrid analysis is performed on f_{viz} (step 4), resulting in the production of the set of f_{viz} -instances whose m_{src} -sub-instances satisfy those constraints.

VLM-sub-instances present in the obtained f_{viz} -instances are then rendered accordingly by the Lightning tool (step 5) to produce a visualization similar to those depicted in Fig. 4.7-4.10.

The visualization of those instances is then shown to the domain expert for validation. If it hinders the comprehension of a given instance, the domain expert can ask f_{viz} to be refined (step 6). If on the contrary the visualization is sufficiently intuitive, the domain expert carries on his review. As soon as an anomaly is found (step 7), it is communicated to the transformation engineer who refines f accordingly (step 8). We note that the visualization of traces can also help the transformation engineer in determining which mapping is faulty. Any structural changes brought during the refinement of f (i.e., changes in the mapping names and types) should be propagated to f_{viz} (step 9). Hybrid analysis can then once again be applied on f_{viz} for the domain expert to validate the changes brought to f by the transformation engineer (back to step 4). If no anomalies are found after reviewing the newly produced instances, the domain expert can request (step 10) another iteration of VBV (on other test cases or using a different scope) or accept the transformation as valid (if a certain degree of confidence has been reached).

In the next section, we show how this process can be applied to validate the CD2RDBMS transformation.

4.5 Application of VBV to CD2RDBMS

In this section, we exemplify the process introduced previously. We show how we validate an F-Alloy implementation of the benchmark CD2RDBMS transformation using VBV.

4.5.1 CD2RDBMS Specification and First F-Alloy Implementation

In this section, we are interested in providing a correct F-Alloy implementation of the CD2RDBMS transformation whose requirements are given in [2]. Compared to the description of the CD2RDBMS transformation given in Section 3.2, the requirements given in [2] encompass the notion of class persistence¹ which, as we will see, might hinder the good understanding of the said requirements.

In the remainder of this section we focus on the validation of the most error prone mapping, namely `association2column`, that defines how associations are to be transformed.

The statements, taken from [2], that are relevant to the implementation of this mapping are the following:

¹It is thus assumed in this section that a field `is_persistent:Bool` is declared in the signature `Class` of Listing 3.1 to represent this detail

```

1  module CD2RDBMS
2  open CD
3  open RDBMS
4
5  one sig CREATE{
6    class2table: Class -> Table,
7    attribute2column: Attribute -> Column,
8    association2column: Association -> Attribute -> Column,
9    association2FKey: Association -> FKey
10 }
11 pred guard_association2column(ass:Association,att:Attribute){
12   ass.src.is_persistent = True and att.is_primary = True and att.~attrs in ass.dest
13 }
14 pred value_association2column(ass:Association , att:Attribute, c:Column){
15   c.dataType = (att.type = STRING implies Text else Number)
16   c.label[0] = ass.name
17   all i:Int| i >= 0 and i <= sub[#, (ass.dest.^(~src.dest)), 1] implies c.label[add[i,1]] = ( c.label[i
18     ].~name.dest != att.~attrs and c.label[i] != none implies c.label[i].~name.dest.~src.name
19     else none)
20   c.label[#, (ass.dest.*(~src.dest))] = att.name
21   c in CREATE.class2table[ass.src].cols
22   ass.dest.is_persistent = False implies c in CREATE.class2table[ass.src].pkey
23   att.~attrs.is_persistent = False or att.is_primary = True implies c in CREATE.association2FKey[ass
24     ].columns
25 }

```

Listing 4.1: Extract of the CD2RDBMS transformation to be validated using VBV

- Rule 1** "... The resultant table (obtained from a persistent class) should contain ... one or more columns for every association for which the class is marked as being the source ..."
- Rule 2** "... for each association whose `dst` is such a class (*i.e.*, for each association whose destination class is non-persistent), each of the classes attributes should be transformed as per rule 3. The columns should be named `name_transformed_attr` where `name` is the name of the association in question, and `transformed_attr` is a transformed attribute ... The columns will be placed in tables created from persistent classes."
- Rule 3** "Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type."
- Rule 4** "Attributes whose type is a persistent class (resp., association whose `dst` is a persistent class) should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named `name_transformed_attr` where `name` is the attributes' name. The resultant columns should be marked as constituting a foreign key, the `FKey` element created should refer to the table created from the persistent class. (The columns will be placed in tables created from persistent classes.)"
- Rule 5** "Attributes whose type is a non-persistent class (resp., association whose `dst` is a non-persistent class) should be transformed to one or more columns as per rule 2. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes."

Listing 4.1 shows the first attempt of the transformation engineer in implementing the `association2column` mapping, namely prior to VBV.

The `guard_association2column` predicate enforces:

Line 14: for each association whose `src` class is persistent (as per Rule 1) and for each primary attribute `att` contained in its `dst` class the creation of a column (according to Rule 4, resp. Rule 5, if the `dst` class is persistent, resp. non-persistent).

The `value_association2column` predicate enforces:

Line 17: the type of the created column to be equivalent to the type of `att` (Rule 3),

Line 18-20: the name of the created column to follow Rules 2 and 4,

Line 21: the placement of the created column in the table representing the `src` class of `ass` (Rule 2 and 4),

Line 22: the created column to become a primary key of its table if the `dst` class is non-persistent (Rule 5),

Line 23: the created column to be part of a foreign key that refers to the table representing `dst` class if `dst` class is persistent and `att` is primary (Rule 4).

With the CD2RDBMS transformation defined, we now show how to validate it using VBV.

4.5.2 Definition of CD2RDBMS_viz

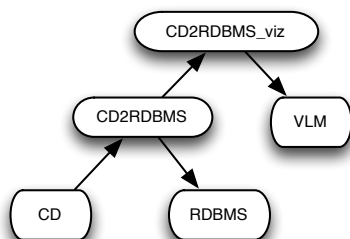


Figure 4.5: Import Hierarchy of CD2RDBMS_viz

According to the process given in Fig. 4.4, when applying VBV to CD2RDBMS the first step is the definition of a visualization for CD2RDBMS. This visualization is expressed, as seen earlier in Sect. 4.4, by a compound f-module called `CD2RDBMS_viz` (depicted in Fig. 4.5) from `CD2RDBMS` to an Alloy module called `VLM` (Visual Language Model) that defines the set of graphical concepts recognized by the supporting tool (*i.e.*, Lightning). To this end, a set of mappings defining the visualization for `CD` and `RDBMS` is first implemented (in `CD_viz` and `RDBMS_viz`, respectively) by the transformation engineer following the domain experts guidance. The set of mappings defining the default visualization of traces (as described in Sect. 4.4) is then added to obtain the final `CD2RDBMS_viz` f-module.

We provide in Listing the `CD_viz` f-module in Listing 4.2 to illustrate how such a visualization is defined.

The mapping `class2Rect` defines that each class is represented by a `RECTANGLE`. This `RECTANGLE` contains a `TEXT`, defined by the `class2Txt` mapping, carrying the name of the class it represents as well as a “persistent” stereotype if that class is persistent. Each


```

module CD_viz

open CD
open VLM

one sig CREATE{
  class2Rect: Class -> RECTANGLE,
  class2Txt: Class -> TEXT,
  attr2Txt: Attribute -> TEXT,
  ass2Connector: Association -> CONNECTOR,
}

pred guard_class2Rect(c:Class){}
pred value_class2Rect(c:Class,r:RECTANGLE){
  r.layout=VERTICAL_LAYOUT
  r.composedOf[0]=CREATE.class2Txt[c]
}

pred guard_class2Txt(c:Class){}
pred value_class2Txt(c:Class,t:TEXT){
  t.textLabel[0]=c.name
  t.textLabel[1]=(c.is_persistent=False
    implies none else "<<Persistent>>")
  t.color=BLACK
  t.isBold=True
}

pred guard_attr2Txt(a:Attribute){
  a.type in PrimitiveDataType
}
pred value_attr2Txt(a:Attribute,t:TEXT){
  t.textLabel[0]=a.name
  t.textLabel[1]=":"
  t.textLabel[2]=a.type
  t.color= (a.is_primary=False implies
    BLACK else RED)
  t in CREATE.class2Rect[a.~attrs].
    composedOf[1]
}

pred guard_ass2Connector(a:Association){}
pred value_ass2Connector(a: Association ,c:
  CONNECTOR){
  c.connectorLabel[0]=a.name
  c.source=CREATE.class2Rect[a.src]
  c.target=CREATE.class2Rect[a.dest]
  c.color=RED
}
    
```

Listing 4.2: The CD_viz f-module defining how CD instances are to be graphically rendered

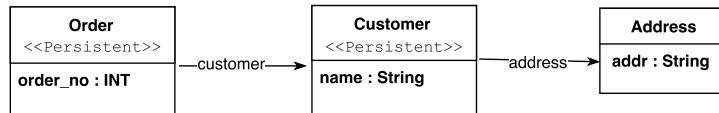


Figure 4.6: Example of Class Diagram that can be given as input to the CD2RDBMS Transformation

attribute is represented by a TEXT in the RECTANGLE representing the class containing it, as defined by the attr2Txt mapping. This TEXT is of the form “att:type” where att is the name of attribute and type its type. Furthermore, TEXTS representing primary attributes are highlighted in red. Finally, the ass2Connector mapping defines that each association is represented by a CONNECTOR from the RECTANGLE representing its source class to the RECTANGLE representing its destination class.

4.5.3 Example of VBV Iteration Using a Specific Test Case

In the first iteration, we use the example provided in [2] as test case. More specifically, the class diagram depicted in Fig. 4.6 is used as input to test the behavior of the CD2RDBMS transformation.

To enforce the use of this input, additional constraints are appended to the original CD module (introduced in Listing 3.1). Those constraints are given in Listing 4.3, where, *e.g.*, one class (Class1) is constrained to be named "Order", to be persistent, and to have a primary attribute named "order_no" of type INT. After constraining the CD module an hybrid analysis on CD2RDBMS_viz is performed. Following Definition 11, the hybrid analysis of CD2RDBMS_viz starts by applying Alloy analysis on CD, yielding the CD-instance depicted in Fig. 4.6. It then performs an F-Alloy interpretation on CD2RDBMS given the CD-instance, yielding the corresponding CD2RDBMS-instance. Finally, it builds a CD2RDBMS_viz-instance from the interpretation of CD2RDBMS_viz given the CD2RDBMS-instance.

```

// ASSOCIATIONS
one sig Ass1 extends Association{
  name="customer"
  src=Class1
  dest=Class2
}
one sig Ass2 extends Association{
  name="address"
  src=Class2
  dest=Class3
}

// CLASSES
one sig Class1 extends Class{
  name="Order"
  is_persistent=True
  attrs=Attr1
}
one sig Class2 extends Class{
  name="Customer"
  is_persistent=True
  attrs=Attr2
}
one sig Class3 extends Class{
  name="Adress"
  is_persistent=False
  attrs=Attr3
}

// ATTRIBUTES
one sig Attr1 extends Attribute{
  name="order_no"
  is_primary=True
  type=INT
}
one sig Attr2 extends Attribute{
  name="name"
  is_primary=True
  type=STRING
}
one sig Attr3 extends Attribute{
  name="addr"
  is_primary=True
  type=STRING
}

```

Listing 4.3: Alloy snippet appended to the CD module to enforce the testcase given in [2]

After hybrid analysis this CD2RDBMS_viz-instance is rendered graphically as shown in Fig. 4.7 and submitted to the domain expert for validation.

The domain expert finds that the transformation is not behaving properly: the columns in the foreign key FKEY do not cover all the primary keys of the Customer table. Communication between domain expert and transformation engineer leads to the conclusion that Rule 4 is ambiguous. More specifically, the use of the terms “primary key attribute” was unclear: the transformation engineer understood from Rule 4 that a column is to be created for each primary attribute of the persistent `dst` class while the domain expert meant that a column is to be created for each primary key in the table corresponding to the persistent `dst` class.

To resolve this error, the domain expert rephrases Rule 4 and the transformation engineer accordingly refines the F-Alloy guard predicate in line 14 as follows:

```

14  ass.src.is_persistent=True and    att.is_primary=True and
    (att in (ass.dest.*(~src.dest & Class ->False.~is_persistent)).attrs)

```

Listing 4.4: F-Alloy Rule Interpretation pseudo code

Columns are now not only created for each primary attribute of the `dst` class but also for all the other primary attributes which would lead to the creation of primary keys in the table corresponding to the `dst` class.

Hybrid analysis is performed again on CD2RDBMS_viz after this modification. This time, the correct instance is produced as shown in Fig. 4.8.

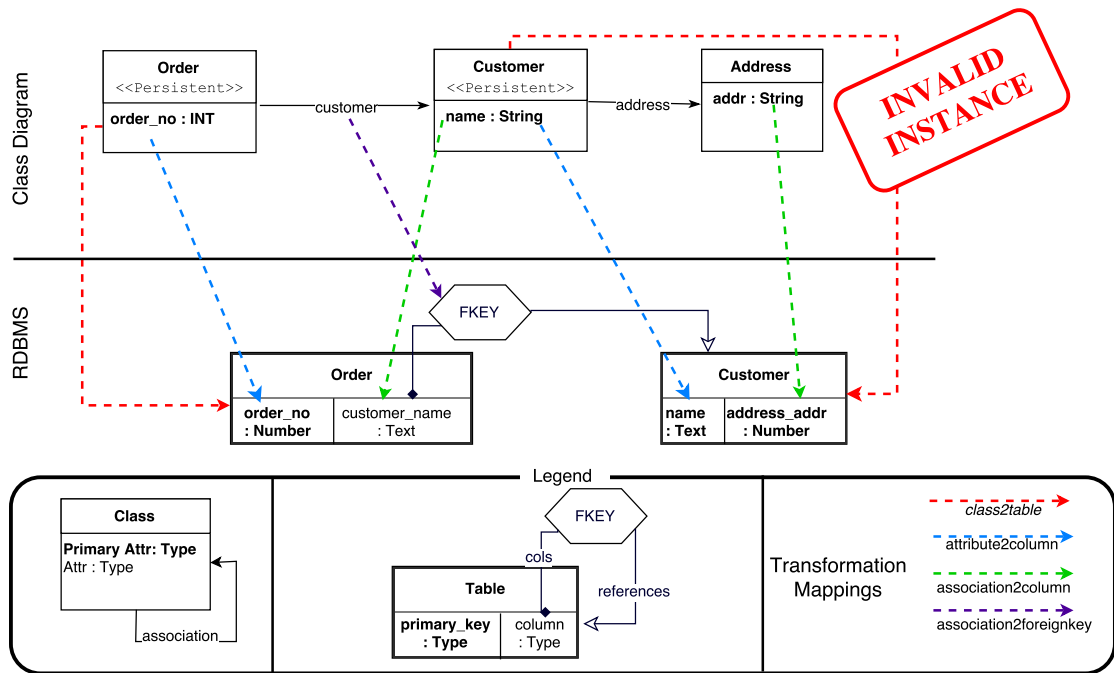


Figure 4.7: Visualization of a bug in the computation of the CD2RDBMS transformation: FK key missing for primary key columns obtained from associations.

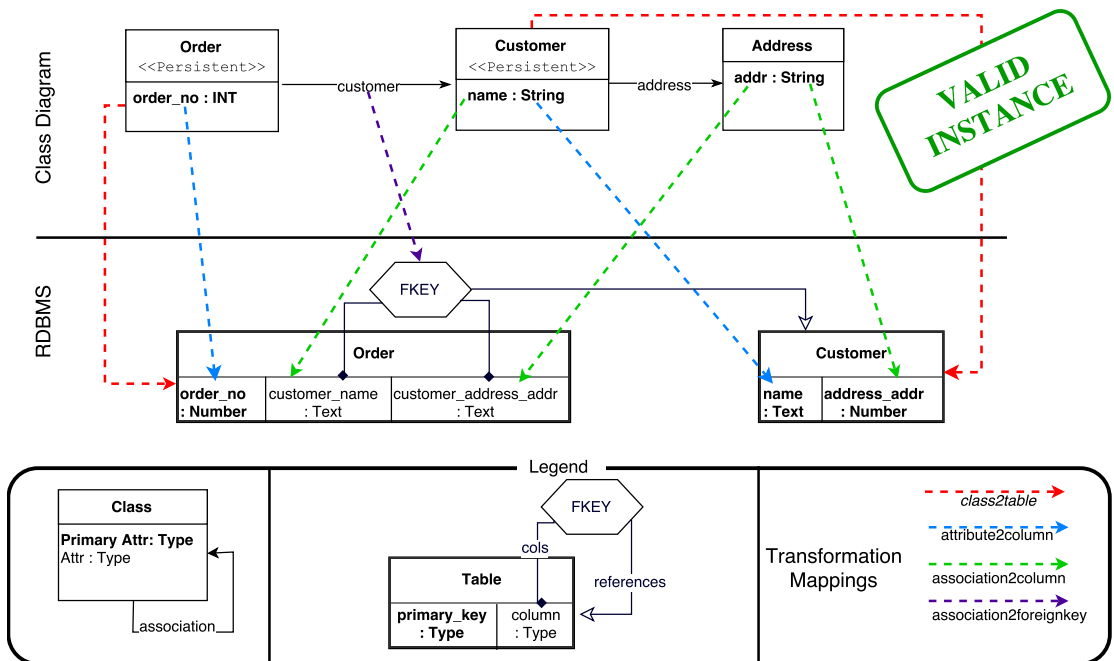


Figure 4.8: Visualization of the case depicted in Fig.4.7 after refinement of the CD2RDBMS transformation.

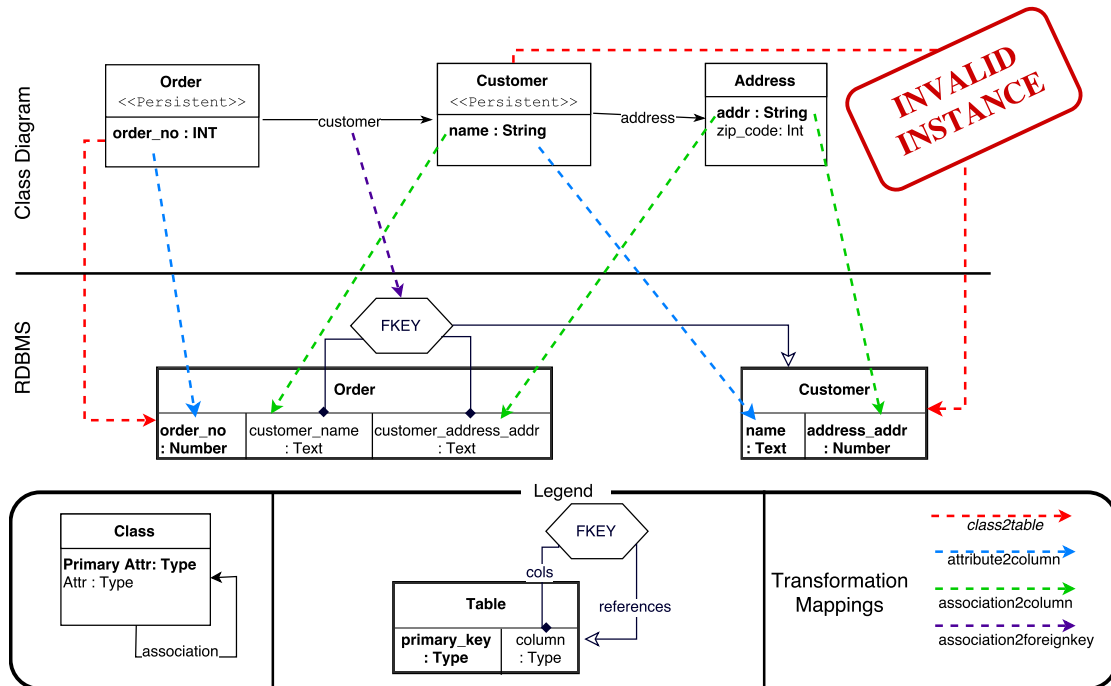


Figure 4.9: Visualization of a bug in the computation of the CD2RDBMS transformation: non-primary attribute of non persistent classes are lost in the transformation

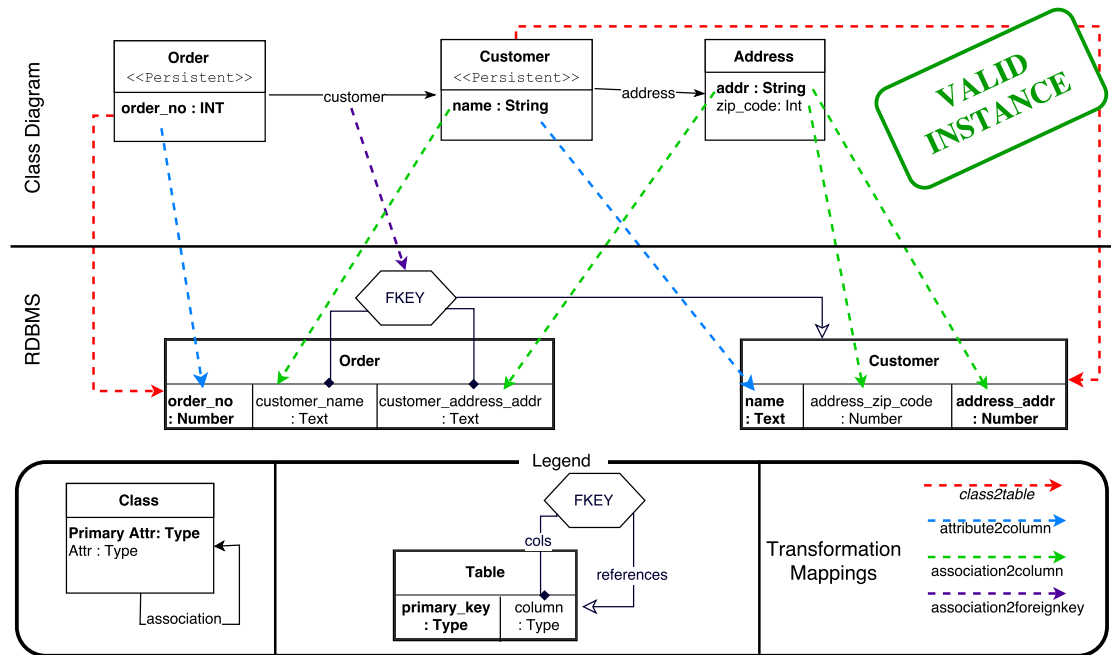


Figure 4.10: Visualization of the case depicted in Fig.4.9 after refinement of the CD2RDBMS transformation.

4.5.4 Example of VBV Iteration Using Random Instance Generation

A second iteration of VBV is then performed. This time, the domain expert requests to add random CD elements to the previous test case to see how the transformation would perform. The transformation engineer hence alters constraints in the CD module to fulfill this request.

A hybrid analysis of CD2RDBMS_viz is then performed and resulting instances are rendered and provided to the domain expert. Figure 4.9 depicts one of those instances containing an anomaly found by the domain expert, namely, there is no column corresponding to the `zip_code` attribute (emphasized in the visualization by the absence of connector originating from `zip_code`).

The lack of corresponding column for an attribute of a non-persistent `dst` class – `zip_code` is an attribute of the non persistent class `Address` which is the destination of the association `address` – is a violation of Rule 2. After discussion, the transformation engineer realizes that he mistakenly applied Rule 4 to the case of non-persistent classes, ignoring completely the directives of Rule 2. To fix this error, the transformation engineer refines the guard predicate `guard_association2column`. This time, two distinct cases are implemented, to cater for the two rules, respectively. This refinement leads to the following modification of line 14:

```

14 (ass.dest.is_persistent=False implies att in ass.dest.attrs)
    and
    (ass.dest.is_persistent=True implies
     (att in (ass.dest.*(~src.dest & Class ->False.-is_persistent)).attrs and att.is_primary=True))

```

Listing 4.5: F-Alloy Rule Interpretation pseudo code

This new version of the guard now checks first whether the `dst` class of association `ass` is persistent or not:

- In case the association `dst` class is persistent, the behavior stays the same as in previous version: a column is created for each primary attribute which would lead to the creation of primary keys in the table corresponding to the `dst` class (Rule 4).
- In case the association `dst` class is not persistent, a column is now created for each attribute of the `dst` class (Rule 2).

Hybrid analysis is performed again on CD2RDBMS_viz after this modification yielding a set of instances which do not reproduce this error.

In this set, the instance containing the same class diagram as the one shown in Fig. 4.9 is shown in Fig. 4.10.

4.6 Evaluation of Hybrid Analysis

Alloy analysis can be used throughout model driven software development processes for validating models, where instances of models are automatically generated and reviewed for potential errors. Such a scenario can be found in case of agile software development where the designer incrementally improves his models after spotting errors in instances obtained by analysis [45].

In this section, we evaluate the efficiency of hybrid analysis by comparing its performance with Alloy analysis on two f-modules introduced in previous sections: the basic transformation from class diagrams to relational databases (i.e., the CD2RDBMS example), and the compound transformation CD2RDBMS_viz that defines a visualization for CD2RDBMS.

Alloy analysis Scope	CD	CD2RDBMS		CD2RDBMS_viz	
	Alloy Analysis	Alloy Analysis	Hybrid Analysis	Alloy Analysis	Hybrid Analysis
5	26	1720	95	3078	229
10	53	7251	181	15862	386
15	76	24671	173	∞	337
20	1844	∞	1918	∞	2053

Table 4.1: Comparative table: Time needed in milliseconds to find the first instance

According to Definition 11, the hybrid analysis of module CD2RDBMS_viz performs as follows:

1. CD2RDBMS_viz being an f-module, hybrid analysis is performed on the CD2RDBMS module.
 - (a) the CD2RDBMS module being an f-module, hybrid analysis is performed on the CD module.
 - (b) the CD module being an Alloy module, CD-instances are generated using the Alloy analyzer.
 - (c) For each instance obtained in step b, an interpretation of CD2RDBMS is performed – resulting in a set of CD2RDBMS-instances.
2. For each instance obtained in step c, an interpretation of CD2RDBMS_viz is performed – resulting in a set of CD2RDBMS_viz-instances.

Note that steps (a) to (c) also correspond to the hybrid analysis of CD2RDBMS.

Time measurements of Alloy and hybrid analysis applied to the CD2RDBMS and CD2RDBMS_viz modules, as well as the time needed for the Alloy analysis of the CD module can be found in Table 4.1 (times given in ms). Note that the symbol ∞ has been used to mark operations that failed to finish. All experiments were carried out on a computer with a Quad-Core Intel i7 CPU and 8 GB memory. Measurements show that the time needed to perform an Alloy analysis increases substantially with the scope and the size of the model. They also show that the time needed to find the first instance of module CD2RDBMS and of module CD2RDBMS_viz using hybrid analysis, is of the same order of magnitude as the time needed to find the first instance of the CD module, for a given scope. These observations provide us with positive support for Proposition 1.

Note that the hybrid analysis of CD2RDBMS and CD2RDBMS_viz surprisingly took less time to complete for a scope of 15 than for a scope of 10. This is due to the fact that the time needed for interpretation to complete is proportional to the size of the instances given to the interpreter and to the fact that the first instance obtained by the analysis of CD with a scope of 10 was bigger than the one obtained with a scope of 15.

4.7 Related Work on Model Transformation Validation

On the Improvement of Alloy Analysis Performances: Our work takes its root in the general problem of speeding up the analysis of Alloy specifications. Different approaches have been proposed for this in the literature. To understand these approaches,

we need to recall how the Alloy Analyzer works. The analysis is based on transforming the Alloy model into a propositional formula that is fed into an off-the-shelf SAT solver.

Alloy analysis can be optimized by improving the transformation from Alloy model to propositional formula. In [97] traditional compiler optimizations are used to improve this transformation, resulting in some cases in time reductions of an order of magnitude.

A different class of optimization techniques is based on applying slicing techniques to Alloy models. In [98] a sub model is identified, called a base slice, that is simpler and more efficiently solvable. Such a base slice can either be proven unsolvable, or extended in a systematic fashion into a full solution. This second approach is closer to our proposed approach: in our case the extension of a partial instance to a complete instance is done via interpretation, while in [98] it is done using a constrained analysis (guaranteeing that a solution for the whole model satisfies the constraints of the base slice).

Yet another type of approach, described in [99], proceeds by annotating Alloy models with meta-information that allows to use domain-specific solvers (e.g., String and Integer solvers) to solve sub-models and combine the output of those solvers with the SAT-based back-end of the Alloy Analyzer.

Rather than improving the speed of the analysis one can attempt to improve its applicability. More specifically, an approach based on SMT solvers allows to drop the finite scope assumption of Alloy's analyzer and actually prove properties of a model regardless of the scope [100]. The drawback of such an approach is the possible need for manual intervention.

On the Verification of Model Transformations: As mentioned in Section 2.4.2 Alloy has been used in the past to verify model transformations. Anastasakis et al. [42] use Alloy to analyze the correctness of model transformations. They resort to their tool UML2Alloy [67] to transform the source and target metamodels into Alloy and translate the transformation rules into mapping relations and predicates at the Alloy level. The goal of their work is to check that the target instances represent indeed models conforming to the target metamodel of the transformation. This is done by checking an Alloy assertion using the Alloy analyzer. In a similar line of work Baresi et al. [66] use Alloy to represent graph transformations represented in the AGG formalism. They use the Alloy analyzer to verify the correctness of the transformation by generating possible traces. We can similarly use Alloy's analysis features to verify model transformations represented in F-Alloy with the added value of being able to use the more efficient hybrid analysis introduced in Section 4.3.

On Visualization Based Validation: In this chapter we proposed an approach to enable the systematic validation of f-modules through the domain specific visualization of a sample of instances. The main incentive to support visualization based validation is to involve domain experts in the process of validation by providing them with material they are familiar with. Validation based on visual feedbacks is not something new, and can be encountered in various other domains [101–104]. As an example, we have seen earlier that the Alloy language comes with a tool, the Alloy Analyzer, allowing the generation and visual rendering of instances as graphs (where atoms are nodes and links are edges). This tool and approach has been successfully applied in the validation of software systems [105, 106], and generally contributed to close the gap between the engineer modeling the system in Alloy, and their client, the domain expert for which the system is designed [107]. However, it has been highlighted in [44] that despite recent advances in Alloy instances representations [108], intuitive visualization of large instances can only be achieved using the knowledge of their domain of application, namely domain specific visualization.

In this work, we proposed to apply the domain specific visualization based validation

approach to the engineering of model transformations. Compared to other domains reported in earlier work, this domain is inherently more complex, (because the validation targets are now transformations themselves, and we need compound transformations from them to VLM to enable the validation process), hence more error-prone to implement. We have seen in Sect. 4.5 that errors can not only come from transformation engineers, e.g., when they failed to respect the transformation specifications written in natural language, but can also come from domain experts, e.g., when they failed to be precise with the intention of the transformations. As a consequence, the involvement of domain experts in the validation process becomes even more necessary. This work is, to our knowledge, the first that thoroughly goes in this direction for validating model transformations. In [109], a visualization technique was proposed to visualize traceability links in (chains of) model transformations, to facilitate the tracking of the origin of errors in transformation chains. However, the absence of domain specific visualization of the source and target models prevents the effective involvement of domain experts in the loop. On the contrary, in our approach, the visualization of traces is not only a way to track the origin of errors in a transformation output, but also together with the domain specific visualization of both the source and target models, it offers additional means to the domain experts to validate the transformation themselves (as seen in Sect. 4.5).

4.8 Summary

In this chapter, we have presented an extension of F-Alloy enabling the expression of compound model transformations, i.e., model transformation specifications whose source or target metamodel are themselves model transformation specifications. We have shown the usefulness of compound model transformations by presenting a new approach to the validation of model transformations entirely based on them.

This novel approach, called VBV (Visualization Based Validation), relies on compound model transformations and on Alloy analysis to produce visual traces to be reviewed by a domain expert.

A requirement critical to the usability of this VBV approach is the efficiency of compound model transformations analysis. To fulfill this requirement, we defined hybrid analysis as a combined use of Alloy analysis and F-Alloy interpretation enabling to reduce the complexity of F-Alloy specification analysis to that of its left-most source module, and have shown semantic equivalence between Alloy analysis and hybrid analysis.

In the next chapter, we present an approach to the specification of DSMLs using exclusively the Alloy and F-Alloy language and present a design process relying on visual feedbacks (like VBV) to enable the seamless validation of the designed DSML, at each iteration.

An Approach Towards Defining DSMLS Using Alloy

In the two previous chapters, we have investigated the use of Alloy in the specification and verification of model transformations. From this investigation came two major contributions, namely (1) F-Alloy, a new Alloy-based language allowing the specification of efficiently computable model transformations and (2) VBV, a new validation approach reducing the gap between formal verification and its main stakeholders: domain experts.

In this chapter, we propose an approach to the definition of domain specific modeling languages based on those two contributions. The chapter is structured as follows. In the first section, we introduce the Structured Business Process language, which we use as case study to illustrate our approach. Section 5.2 presents an approach to the definition of languages in Alloy. An agile design cycle (consisting of small iterations) allowing to validate each component of the language with the help of the domain expert is presented in Section 5.3. Finally, we discuss how our approach compares to other attempts to use Alloy in the definition of DSLs in Section 5.4 before concluding the chapter with a summary of the contribution.

5.1 The Structured Business Process Case Study

In this part, we illustrate language design with the help of a concrete language, the Structured Business Process (SBP) language [110].

Structured business processes consist of *tasks* representing actions performed towards the completion of the process and of *control nodes* structuring the process. Those tasks and control nodes are interconnected using transitions so that the following holds:

1. The process has a unique start and end, represented by the Start and End control nodes, so that no transition is incoming to Start or outgoing from End.

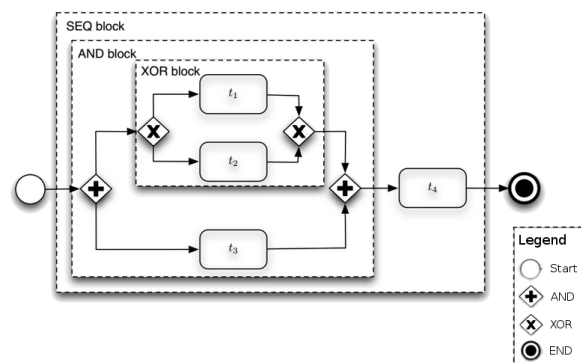


Figure 5.1: A Structured Business Process

2. Each task has exactly one incoming and one outgoing transition.
3. XOR and AND are control nodes used to delimit blocks representing the nesting of processes. The difference between XOR and AND is purely semantical: while AND means that all sub-processes (outgoing transitions) need to be processed, XOR specifies that exactly one of them has to be processed.
4. XOR and AND control nodes have one incoming and more than one outgoing transition if they are used to open a new block (in which case they are called XOR split and AND split), or more than one incoming and one outgoing transition if they are used to close a new block (in which case they are called XOR join and AND join).
5. A block opened by an AND split or XOR split needs to be closed by an AND join or XOR join, respectively.
6. The process is acyclic (all tasks are traversed at most once).

An example of a business process expressed in this language is given in fig. 5.1 (adapted from [111]). The representation used in this figure relies on traditional notation from the business process community.

This choice of case study is based on the fact that:

- The SBP's specification has been formalized in [111], thus providing a precise description of the syntax and semantics of this language.
- It has sufficient complexity to illustrate the usefulness of our tool.
- It is practically relevant since many existing business processes are expressible in this form [111].

5.2 An Alloy-based Language Definition

As defined in Section 2.2.1, a domain specific modeling language definition is composed of three components, namely abstract syntax, concrete syntax and semantics. In this section, we detail how each of those components can be defined using Alloy.

An overview of the proposed approach is given in Figure 5.2.

5.2.1 Abstract Syntax

We have seen in Section 2.2.1 that the abstract syntax of a language aims at defining the set of valid language model in terms of concepts, relations, and constraints. The Alloy language being perfectly suited to define such structure, we propose in our approach to define the abstract syntax of languages by an Alloy module. We call this module *Abstract Syntax Model (ASM)*.

As the abstract syntax aims at defining the set of valid language models, we define language models as follows:

Definition 12 (Language model). *Given a language \mathcal{L} , we call language model of \mathcal{L} any instance of \mathcal{L} 's abstract syntax model.*

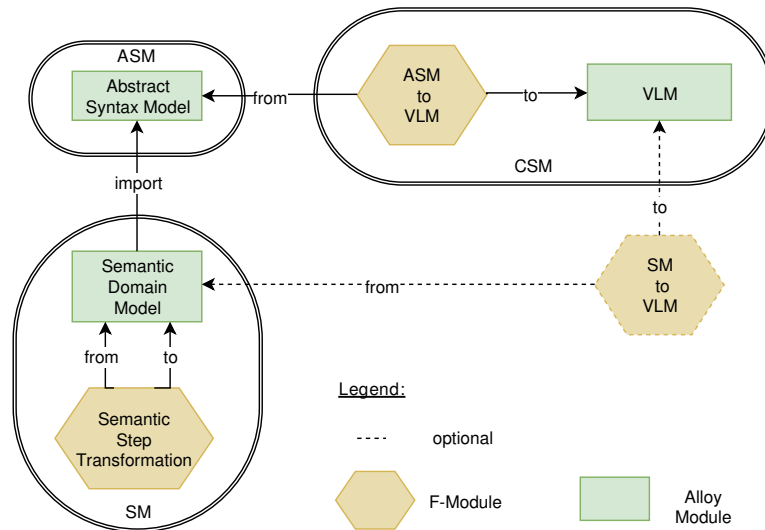


Figure 5.2: Overview of an Approach to the Alloy-based Definition of Domain Specific Modeling Languages

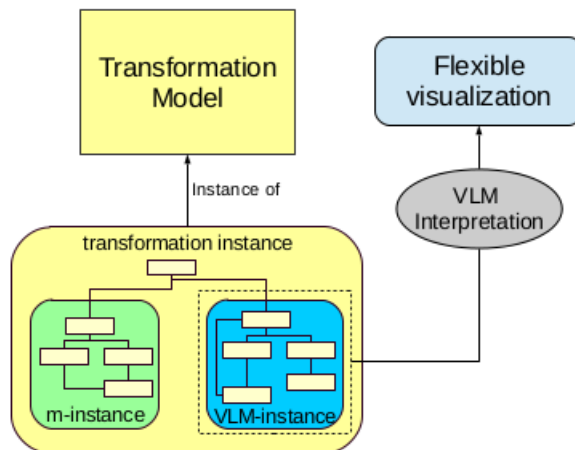


Figure 5.3: Overview of our approach to the definition and rendering of a language's concrete syntax

5.2.2 Concrete Syntax

We recall from Section 2.2.1 that the concrete syntax of a language defines the domain specific representation of language models. It is commonly understood that this definition can be used to:

1. visualize given language models.
2. create and edit language models.

In the present approach, we do not consider the creation and edition of language models and thus assume that the concrete syntax solely aims at defining a domain specific visualization. This choice is motivated by the fact that our primary motive is the intuitive validation of DSML specifications for which only the visualization is necessary. Moreover, we will see in Chapter 6 that visualization can be used to ease the edition of language models when used in combination with a non-graphical editor. Finally, we believe that our approach can be extended to also provide editor support the moment F-Alloy enables the expression of bidirectional transformations.

In Section 4.4, we have already presented an approach to the definition of model and model transformation's domain specific visualizations. This approach consists in defining an F-Alloy model transformation from the language model (ASM-instance) to be visualized, to a VLM-instance. In this work, we have defined our own VLM (given in Annexe D), and have developed a tool called Lightning, introduced in the next chapter, which enables the visual rendering of VLM-instances. This approach to the definition of a concrete syntax (depicted in Figure 5.3) follows the approach that Kleppe describes in [41].

5.2.3 Semantics

We have seen in Section 2.2.1 that semantics aims at providing meaning to language models, and that there exists four main approaches to its definition.

We choose to focus on the definition of operational semantics. This choice is motivated by (1) the fact that the mathematical objects composing a denotational semantics definition can hardly be presented in an intuitive form to domain expert, by (2) the fact that pragmatic semantics validation is naturally performed by domain experts when they get hold of the tooling derived from the DSML definitions, and by (3) the fact that translational semantics simply consists in defining a transformation between two abstract syntax models, which is already supported by F-Alloy.

Defining the operational semantics of a language consists in defining an abstract state machine whose execution on a given language model gives its meaning. The definition of such an abstract state machine consists in:

- A *semantic domain model* defining the notion of semantic state (state of the abstract state machine). The semantic domain model is given as an Alloy module importing the ASM (abstract syntax model) so as to define the concept of semantic state reusing concepts of the ASM.
- A *semantic step transformation* defining, given a semantic state (semantic domain model instance), how to obtain the next semantics state. The semantic step transformation is thus defined as an F-Alloy endogenous model transformation from/to the Semantic Domain Model.

Each semantic domain model instance representing a semantic state, it is necessary for visualization-based validation purpose to define a domain specific visualization of informations relative to the semantic state. A semantic domain model is thus in generally accompanied by its own concrete syntax definition, taking the form of a f-module transformation, from the semantic domain model to the VLM. The *semantic model (SM)* of a language \mathcal{L} consists thus of the semantic domain model, its concrete syntax, and the semantic step transformation.

5.3 An Agile Design Cycle

We now define an agile design cycle to be followed when defining languages as introduced in Section 5.2, and illustrate its use with our SBP case study.

We note that two actors are involved in this design cycle:

- The *Domain Expert* is acquainted with the language to be designed. He has strong expectations on how language models are to be represented and executed.
- The *Language Engineer* is an Alloy and F-Alloy expert who follows our approach to define a language, following her/his understanding of the language specification.

This cycle, depicted in Fig. 5.4 is “agile” in the sense that it consists of very short iterations followed by validations, hence allowing to track down design errors at the earliest stages.

5.3.1 Designing the Abstract Syntax

The Language Engineer starts the design of the SBP language by defining its ASM (ASM playing a central role in the language definition). To do so, (s)he designs an Alloy module from the specification of the language given in Section 5.1. We give in Listing. 5.1 an excerpt of the ASM designed.

After producing this first version of the ASM, the language engineer performs an Alloy analysis to validate that the Alloy specification conforms to the requirements. This Alloy analysis yields a set of valid instances (one is given in Figure 5.5), that can be used by the language engineer to evaluate the correctness of her/his ASM specification. Designing the ASM and validating it using Alloy corresponds to the cycle labeled 1 in Fig. 5.4.

Validating an ASM specification by reviewing instances graphically rendered by the Alloy analyzer can be a tedious exercise due to the fact that those representations of

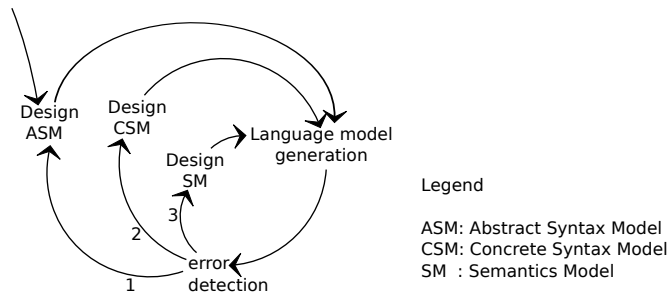


Figure 5.4: Spiral diagram depicting how languages defined following the proposed approach can be incrementally designed

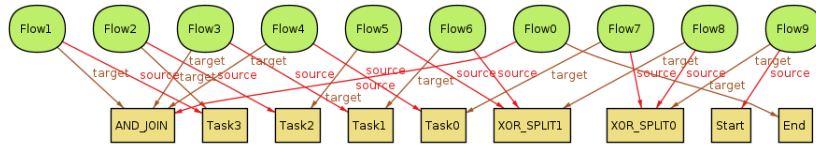


Figure 5.5: Raw visualization of a language model (using Alloy's Magic Layout)

```

abstract sig Node{}
abstract sig Control extends Node{}
one sig Start extends Node{}{this not in Flow.target}
one sig End extends Node{}{this not in Flow.source}
sig Task extends Node{}
sig AND_JOIN, AND_SPLIT, XOR_JOIN, XOR_SPLIT extends Control{}
sig Flow{
  source: Node,
  target: Node
}
fact acyclic{
  all n: Node | n not in n.^((~target).source)
}

```

Listing 5.1: excerpt of the SBP's abstract syntax model

instances are direct reflections of the abstract syntax model structure (as imagined by the language engineer). More precisely:

1. The lack of variations in the way concepts are represented (all are nodes of a graph) leads to an unnecessary verbose representation that can quickly become hard to process [53].
2. The fact that the language model representation is heavily influenced by design choices made by the language engineer (as they follow the structure of the ASM produced) hinders the efficient involvement of the domain expert in the validation process.

To improve the readability of the generated instances (1) and to allow the involvement of the domain expert in the validation process (2), it is thus necessary for the language engineer to proceed to the concrete syntax design.

In a nutshell, proceeding to the concrete syntax design, cycle **2** in Figure 5.4, is advised once at least one of the following conditions holds:

- the language engineer is, after multiple round of Alloy analysis, confident of his ASM design.
- the instances yielded by the Alloy analyzer are too complex to be used as a validation mean.
- the language engineer is uncertain about his implementation of a given requirement and would like to consult the domain expert.

5.3.2 Designing the Concrete Syntax

The concrete syntax design consists, as seen in Section 5.2 in defining a model transformation from the previously defined Abstract Syntax Model (ASM) to a predefined visual language metamodel (VLM), similarly to the definition of a domain specific visualization as defined in Section 4.4.

```

/* Each task is represented by a rectangle, and each node has its corresponding label */
one sig CREATE{
  mapTask: Task -> RECTANGLE,
  mapNodeText: Node -> TEXT
}
// Each Task is represented by a white-filled rectangle containing its name
pred guard_mapTask{}
pred value_mapTask(n:Task, r:RECTANGLE) {
  r.layout = VERTICAL_LAYOUT
  r.color = WHITE
  r.composedOf[0] = CREATE.mapNodeText[n]
}

// Each node is mapped to a label representing its name in black without style
pred guard_mapNodeText{}
pred value_mapNodeText(n:Node, t:TEXT) {
  t.color = BLACK
  t.isItalic = False
  t.isBold = False
  t.textLabel[0] = n
}

```

Listing 5.2: excerpt of the SBP to VLM transformation

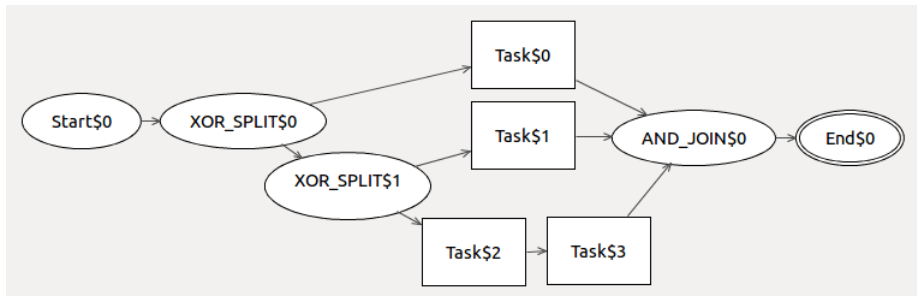


Figure 5.6: Visualization of the instance depicted in fig. 5.5 using its concrete syntax definition

In Listing 5.2, we provide an excerpt of such a transformation defining how tasks are to be rendered.

Once such a concrete syntax is defined, it becomes possible to provide a domain specific visualization to each generated instance and thus to involve the domain expert in the design validation. We provide in Figure 5.6 the representation, using the concrete syntax defined by the language engineer, of the language model previously shown in fig. 5.5. Through this visualization, it is obvious to the domain expert that the language specification is erroneous as two XOR splits cannot possibly be converging to a single AND join. After discussions with the domain expert, the language engineer introduces a new concept (s)he arbitrarily calls control box aiming at pairing matching split and join nodes together. This fix is given in Listing 5.3.

Repeating the instance generation after this modification shows that the introduction of this new concept has indeed corrected the design error previously identified. The error processing we just presented illustrates a transit from cycle **2** to cycle **1** in fig. 5.4, i.e., to the case where an error found in the visualization reveals an error in the underlying abstract syntax model. Of course the transformation model describing the visualization may be faulty itself. In this case the error in the visual representation may point to an error in the concrete syntax model. This situation corresponds to another iteration of

```
sig ControlBox {
  split: Control,
  join: Control
}{
  // SPLIT AND JOIN HAVE SAME NATURE
  (split in AND_SPLIT and join in AND_JOIN) or (split in XOR_SPLIT and join in XOR_JOIN)
  // PAIRING EACH SPLIT WITH A GIVEN JOIN
  all s: (succ[split]) | s in (preds[join]) and all j: (pre[join]) | j in (succs[split])
}
```

Listing 5.3: excerpt added to the ASM by the language engineer to enforce well formedness of split and join nestings

cycle **2**, leading to a refinement of the concrete syntax design.

The language engineer starts working on defining the operational semantics of the language once the domain expert is satisfied with the defined concrete syntax.

5.3.3 Designing the Operational Semantics

We recall from Section 5.2 that defining an operational semantics consists in defining an abstract state machine whose execution on a given language model defines the language model's meaning. In our approach, this abstract state machine definition consists of two artifacts: the semantic domain model and the semantic step transformation.

Designing the Semantic Domain Model

Defining the Semantic Domain Model consists in answering the following questions:

1. If a language model would be executed, what would characterize a state of execution?
2. What would then be the first state of execution?

A quick discussion with the domain expert allows the language engineer to ascertain the answer of those two questions for the SBP language: (1) An execution state consists of a set of active nodes, and (2) the first state of execution should have as single active node the start node.

The semantic domain of the SBP language is thus implemented accordingly as given in Listing 5.4.

Designing the Semantic Step Transformation

Once the semantic domain model is defined, the language engineer proceeds to the definition of the semantic step transformation.

The semantic step transformation is an endogenous transformation from/to the semantic domain model aiming at defining valid state transitions. The language engineer thus has to answer the following two questions before implementing the transformation:

1. How to compute, given a state of execution, the valid subsequent state?
2. When does the execution end?

The language engineer, after consulting the domain expert, provides the following answers:


```

module SBP/Semantics/Semantics
open SBP/AbstractSyntax/ASM

one sig State{
  activeNodes: set Node
}
pred init [s:State] {
  s.activeNodes = Start
}

run init
    
```

Listing 5.4: the semantic domain model of the SBP language

```

1 module SBP/Semantics/Step
2 open SBP/Semantics/SDM
3
4 one sig UPDATE{
5   map: State -> State
6 }
7 pred guard_map(s: State) {
8   not s = End.(~activeNodes)
9 }
10 pred value_map(s1: State, s2: State) {
11   s2.activeNodes = nextActiveNodes[s1.
12     activeNodes]
    
```

Listing 5.5: Semantic Step Transformation of the SBP language as defined by the language engineer

```

1 fun nextActiveNodes(n: set Node): set Node {
2   //return nextNodes in general except :
3   nextNodes[n - AND_JOIN - XOR_SPLIT] +
4   //if and_join: wait no nodes in n are preceding it
5   nextNodes[{x: n & AND_JOIN | n & predecessors[x]=none}] +
6   //if xor_split: return solely one successor .
7   {x: Node | some y: n & XOR_SPLIT | x= order/max[successors[y]]}
8 }
9
10 fun nextNodes (n: set Node) : set Node { n.(~source).target }
11 fun successors (n: set Node) : set Node { n.^((~source).target) }
12 fun predecessors (n: set Node) : set Node { n.^((~target).source) }
    
```

Listing 5.6: Function computing the set of next active nodes expected in the execution of an SBP language model given a set of active nodes

1. given a set of active nodes, the set of next active nodes can be computed by following the flows and by respecting the following two rules: (1) an AND join node should stay active until no preceding nodes are active and (2) successors of an XOR split node cannot be active at the same time.
2. the execution stops once the end node is reached.

Following those answers, the transformation given in Listing 5.5 is defined.

The transformation updates the `activeNodes` value of the execution `State` only if the `End` node is not contained in the set of `activeNodes` (line 8). The new set of `activeNodes` (line 11) is computed by the function `nextActiveNodes`¹ given in Listing 5.6 following the answer previously formulated.

The operational semantics is now clearly defined, but to be able to visualize an actual execution of an SBP language model, it is necessary to define how active nodes are to be visualized.

Visualizing Semantics

To enable the visualization of an SBP execution as defined by the previously presented operational semantics, the language engineer has to define a transformation from the

¹This function is to be added to the semantic domain model as such a helper function can't be declared in syntactically valid F-Alloy specifications as enforced by the F-Alloy syntax defined in Section 3.3

5.4. RELATED WORK ON DSML ENGINEERING APPROACHES

```

pred value_mapTask(n:Task, r:RECTANGLE) {
  r.layout = VERTICAL_LAYOUT
  r.color = (n in State.activeNodes implies GREEN else WHITE)
  r.composedOf[0] = Bridge.mapNodeText[n]
}

```

Listing 5.7: updated version of the mapTask mapping allowing to render semantic domain specific informations

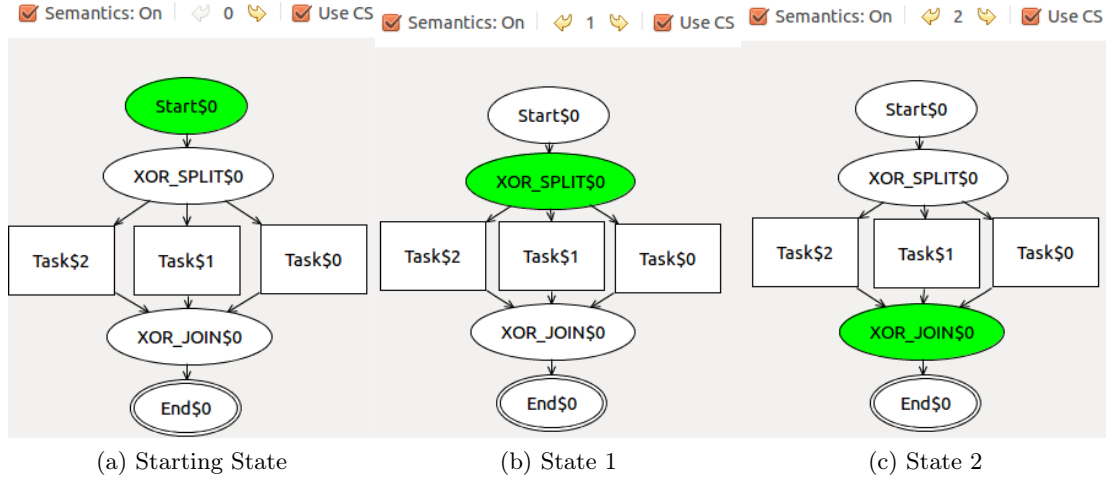


Figure 5.7: Faulty semantics execution of an SBP language model containing XORs

Semantic Domain Model to the Visual Language Metamodel. This transformation generally reuses mappings of the transformation used as a concrete syntax (as the semantic domain model imports the abstract syntax) and just contains additional rules mapping those execution state related properties to a proper visualization. The language engineer decides to differentiate active nodes from others by highlighting them in green. This is achieved by refining the concrete syntax transformation as illustrated in Listing 5.7. An execution of the operational semantics defined can, thanks to this newly defined transformation, be rendered graphically as depicted in Fig. 5.7. This execution visualization reveals an error since one of the task nodes needs to become active between the activation of XOR_SPLIT and XOR_JOIN.

More precisely, the visualization allowed to uncover a small error in the `nextActiveNodes` function line 7 of Listing 5.6, where a call to the `successors` function (yielding the set of all nodes preceded by the ones given in parameter) is used instead of the `nextNodes` function (yielding the set of nodes directly preceded by the ones given as parameters). Repeating this cycle after accordingly replacing the erroneous call to the `successors` function by a call to the `nextNodes` function shows that the bug has been fixed.

5.4 Related Work on DSML Engineering Approaches

In this section, we discuss related work connected to the DSML engineering approach presented in this chapter.

5.4.1 Alloy-based DSML Engineering Approaches

With the emergence of DSML engineering as a popular research topic and the already numerous successful applications of Alloy in the verification and validation of models, investigations on the use of Alloy in the validation and verification of DSML engineering took place. We put here our approach to DSML engineering into perspective with those investigations.

In [112], authors unveil the potential of Alloy in the verification of DSML designs. They define in Alloy modules the abstract syntax and operational semantics of the Maze Game DSML [113]. Verification is then performed by checking assertions on those modules. This verification technique, being natively supported by Alloy, is applicable to DSMLs specified following the approach we presented in this chapter. It is to be noted though that f-modules cannot yet be verified using assertions without being translated first in an Alloy module (following the procedure defined in Section 3.4.2). Hence in our case, the verification of a DSMLs operational semantics using assertion checking requires the semantics step transformation to be translated in Alloy and relevant assertions to be added in the resulting Alloy module. Our approach presents two advantages when compared to the one presented in [112], namely (1) our DSML definition also takes into consideration concrete syntax which can be used to ease the validation of the abstract syntax and semantics and (2) the use of F-Alloy instead of Alloy in the specification of the operational semantics makes semantics execution less time consuming and hence more scalable.

Work in [37] shows that Alloy analysis can play a central role in the iterative (agile) specification of a DSML's abstract syntax. The authors provide empirical evidence that for a given DSML specification, it is possible to identify missing well formedness constraints from the simple observation of generated instances. Our approach extends this practice to all the components of a DSML definition. Work in [37] corresponds precisely to the first cycle of the agile design process we provide in fig.5.4 which was first published in [45] (a year before [37]).

5.4.2 Example-Based Validation Approaches in the OMG World

One of the greatest benefits retained from Alloy in our approach is the ability to generate instances from a DSML design. In the following, we introduce other methodologies relying on instance generation to validate specifications. We then present DSML engineering techniques relying on those methodologies and compare them, when applicable, to the DSML engineering approach we presented in this chapter.

In the OMG world, *UML* [114] and *OCL* [115] constraints can be used to define structures and well formedness rules, respectively. The modeling environment *USE* [116] was designed to enable the validation of specifications expressed in *UML/OCL*. To do so, it relies on the generation of so called snapshots using *SAT*-solvers [117], hence following the same approach than Alloy. The difference with Alloy is that snapshots can also be constructed manually, and that *USE* provides a small DSL called *ASSL* [118] enabling to list modifications to be brought to a given snapshot. The execution of an *ASSL* procedure would then return those snapshots resulting from applying a selection of those modification, so as to still conforming to the metamodel.

USE hence paved the way to instance-based validation support to the numerous already existing *UML* tooling [119] and to existing approaches to DSML engineering relying on *UML* ([120, 121]).

In a very recent work, *USE* has been successfully used in a modeling environment called *metaBest* [122] to validate DSMLs. *MetaBest* is an eclipse-based framework [123]

whose central tool, metaBup, enables the definition of DSMLs following a bottom-up approach [124]. MetaBup takes as input graphical sketches of language models produced by domain experts, in which every graphical element are consistently labeled. From these sketches are derived both the abstract syntax and concrete syntax of the DSML used to express the language models. Models of those syntaxes can then be viewed and corrected if needed. MetaBest also provides two home-brewed DSLs, mmUnit [125] and mmSpec [126] to define well-formedness constraints. The first one, mmUnit, is to be used in combination with sketches representing invalid language model. It allows to textually define why the invalid language model sketched is indeed invalid. From those mmUnit specifications can be derived constraints that should be satisfied in any language model. Those constraints are expressed in the mmSpec language. MmSpec is less expressive than OCL and Alloy but is assumed to be more intuitive. The semantics of mmSpec is defined as a translation to OCL.

The metamodel obtained by metaBup along with the translation of mmSpec constraints translated to OCL can be given as input to USE to generate instances for validation's sake [127]. ASSL being judged too complex and platform dependent to fit in the metaBest framework, yet another language called mmXtens is designed, aiming at defining properties that should hold in instances generated using USE.

To summarize, metaBest can be used to define the abstract and concrete syntax of a DSML following a bottom up approach, which is a drastically different approach than the one we proposed. Yet, the validation of the DSML specification thus obtained can be validated through instance generation, each instance generated being rendered using the concrete syntax of the language. This DSML validation technique is similar to the one we propose. Semantics definition seems to be out of metaBest's scope, yet the abstract syntax model being exportable to UML or Ecore, one can imagine that a variety of tooling [128,129] can be used to define it, the only drawback being that in case of an operational semantics definition, the concrete syntax defined using metaBest cannot be directly reused to visualize semantics execution.

5.4.3 On Concrete Syntax Definitions

In the last decades, multiple works addressed the problem of defining graphical concrete syntaxes. In the following, we present the two most prominent approaches to their definition.

The first approach consists in associating to each concept of the abstract syntax a graphical template. This approach, adopted by Atom3 [130], MetaEdit+ [131], GME [132] and DOME [133] for its simplicity, comes with a major disadvantage. The approach indeed induces structural dependencies between the abstract syntax and the defined concrete syntax. This tends to push the language designer to violate separation of concerns as each abstract syntax concept has to be representable by one graphical construct and vice-versa.

The second approach alleviates this limitation by enabling the language engineer to define relations between abstract syntax concept and their representations as a model transformation from the abstract syntax to a visual language metamodel. This is the case, e.g., of GenGed [134], defining concrete syntax by a TGG model transformation, and of our approach in which concrete syntax is defined as an F-Alloy model transformation. Though separation of concerns is no longer a problem following this approach, using model transformations to define concrete syntax is generally shun as language engineers are assumed to lack expertise in model transformation engineering. In our case, language engineers are assumed to be Alloy specialists as the proposed DSML engineering approach

revolves around the use of Alloy. The F-Alloy language being by definition inherently close to Alloy, this limitation is hence not applicable to our approach.

5.5 Summary

In this chapter, we demonstrated that the abstract syntax, concrete syntax and operational semantics of a DSML can be expressed as a combination of Alloy modules and F-Alloy transformations. We have also shown that DSMLs can be defined following an agile design process in which analysis can be applied at each iteration for validation's sake. We illustrated this design process through the definition and validation of the SBP (Structured Business Process) language.

In the next Chapter we present a tool named Lightning, implementing the approach to DSML engineering we presented in this chapter.

Chapter 6

The Lightning Language Workbench

In this chapter we present the Lightning language workbench, a tool implementing the approach previously introduced in Chapter 5 to enable the agile design of DSMLs. In Section 6.1, we provide an overview of the tool and of its major features. A successful application of the Lightning tool is then given in Section 6.2, where a Robotic DSML has been designed and used. Finally, we conclude this chapter by comparing Lightning to existing language workbenches in Section 6.3 and by summarizing the contributions of this chapter in Section 6.4.

6.1 Tool Presentation

The Lightning language workbench is distributed as an Eclipse plug-in so as to ease future integration with already existing model driven techniques.

Fig.6.1 gives an overview of the tool's graphical user interface. In the following we rely on this figure to introduce Lightning's properties and features.

6.1.1 Lightning Languages

Lightning's primary purpose is to help in the definition of languages following the approach presented in Chapter 5. In the Project Explorer view (Fig. 6.1 left), we can see that a language definition in Lightning takes the form of a directory composed of the following four sub-directories:

AbstractSyntax : contains the set of Alloy modules used in the definition of the abstract syntax of the language. One Alloy module can be marked¹ as the *ASM* of the language with effect that language model are instances of the marked ASM. The ASM module may import other modules in this directory.

Semantics : contains Alloy modules defining the semantic domain of the language and f-modules defining valid semantic steps. A given Alloy module should be marked¹ as the *SDM* of the language (semantic domain model) and an f-module should be marked¹ as the *SST* of the language (semantic step transformation) in order to enable the simulation of a language model execution by (1) obtaining the first state of execution by analyzing the SDM and by (2) advancing step by step in the execution by repeatedly executing the SST transformation.

ConcreteSyntax : contains the set of f-modules defining visualizations of ASM-instances and SDM-instances. It also contains the default *LightningVLM*² Alloy module,

¹through the right click context menu

²This LightningVLM module is automatically added to the concrete syntax directory during the language creation

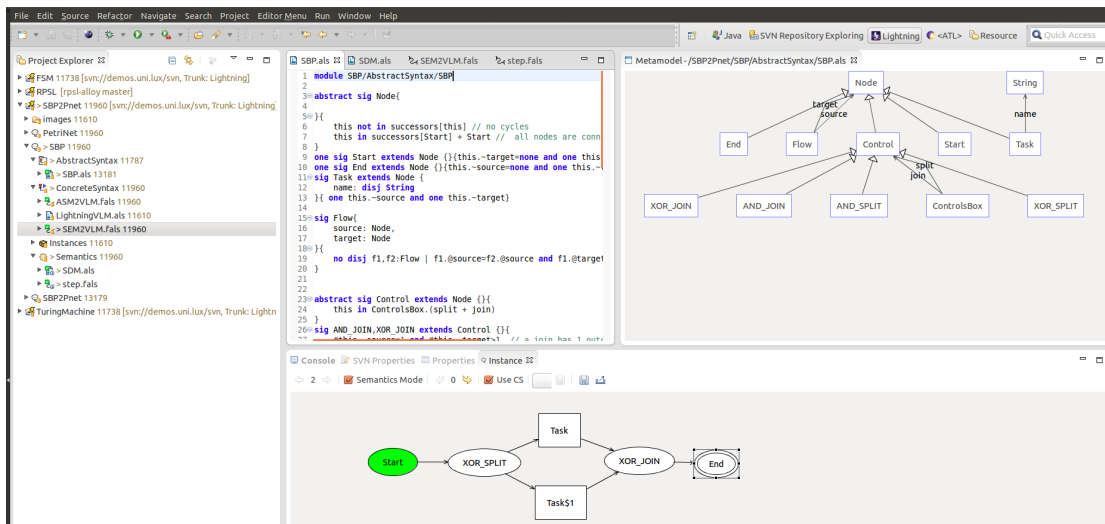


Figure 6.1: An overview of the Lightning Graphical User Interface

declaring the set of visual constructs whose rendering is supported by the tool. In this directory, an f-module should be marked as *CSM* to enable the visualization of instances of selected ASM. An f-module should also be marked as *semantics CSM* to enable the visualization of instances of the selected SDM.

Instances : is the default location in which instances of a language can be saved for future uses. Saved language models can be visualized if a CSM is selected, executed if an SDM and SST are selected or even edited as presented in the next subsection.

6.1.2 Editor

The Lightning tool provides its users with an Alloy and F-Alloy textual editor. An Alloy editor is depicted in the center of Fig.6.1. Those editors are paired with a metamodel view (Fig.6.1, right) providing an overview of the model designed and so that any structural change¹ introduced through the editor is rendered instantly.

Those editors also provide:

- syntax highlighting, coloring keywords in blue and comments in green,
- an outline view, depicted in Fig.6.2, enabling easy access to anything declared in the edited module,
- syntax checking, allowing to ensure that any Alloy module and f-module specified is syntactically correct,
- syntax error marking, enabling to locate which part of the specification is not syntactically correct. The part will be underlined in red, and in some cases, e.g., F-Alloy guard or value predicate missing, quick fixes will be provided (e.g., auto-generation of missing predicate following types in the mapping declaration).

6.1.3 Instance Viewer

From the moment the *ASM* of a language is defined, it is possible to generate language instances using the execution button (▶). Those instances will be available for browsing

¹changes impacting signatures and fields (and not constraints)

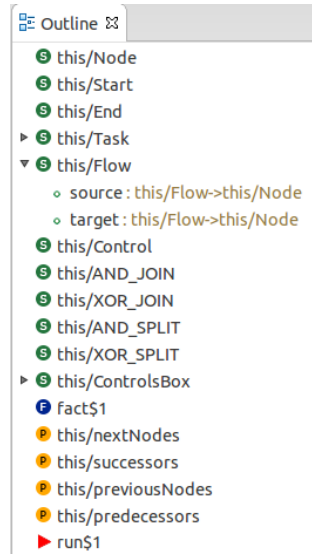






Figure 6.2: Outline view associated to the Alloy editor (currently editing the SBP Alloy module). Clicking an entry of this outline will have as effect to highlight where it has been declared in the editor.

in what we call an instance viewer (See Fig.6.1, bottom part). The instance viewer comes with his own tool-bar allowing the user to perform various actions on the instance displayed. We list here elements composing this tool-bar:

- The instance viewer displays a single language model at a time. If several language models are available for display, it is possible to browse through them using the first set of arrows (← 1 →). The pointer between those two arrows shows the index attributed to the language model currently visualized. The left and right arrow are disabled when there is no more preceding and succeeding language models, respectively.
- Toggling on the semantics mode (**Semantics Mode**) will have as effect to display the first state of execution of the language model displayed. The instance denoting this first state of execution is automatically obtained by:
 1. An automatic refinement of the *ASM* consisting in the addition of constraints so that the only instances obtainable by analyzing the refined ASM is the language model displayed.
 2. An Alloy analysis of the language's *SDM* importing the refined ASM.

We note that activating the semantics mode locks the browsing of language instances and activates the otherwise locked semantic steps executions.

- If the semantics mode is active, then it is possible to execute steps of the semantics on the language model visualized using the second set of arrows (↩ 1 ↪). The right arrow executes a step as defined by the *SST* of the language. The left arrow has as effect to “undo” already performed steps. The index displayed between the two arrows allows to keep track of the number of steps that has been performed in order to reach the semantic state currently visualized.

- At any time, it is possible to configure the instance viewer to use the concrete syntax of the language to view language models and semantic states. This is done by toggling the `use CS` checkbox ( Use CS). This has the following effects:
 - If the semantics mode is disabled: the *CSM* (transformation from *ASM* to *VLM*) is applied on the language model displayed so as to obtain a *VLM*-instance. This *VLM*-instance is then automatically processed by the Lightning tool in order for the instance viewer to display the language model through its domain specific visualization.
 - If the semantics mode is enabled: the *Semantics CSM* (transformation from *SDM* to *VLM*) is applied on the execution state displayed so as to also obtain a *VLM*-instance. Again, this is later processed and rendered graphically by the Lightning tool.
- It is possible to transform a language model into a model of another language (e.g., CD to RDBMS, SBP to PetriNet, ...) by selecting a transformation in the transformation drop-down list (). By default this drop-down list contains all the f-modules having as source the *ASM* of the language model displayed and as target the *ASM* of another language. It is possible to apply multiple transformations at once (e.g., Ecore to CD and CD to RDBMS) and to memorize the chain of transformation using the *save* button next to the drop-down list. Saving a transformation chain has as effect to make the chain directly accessible in the transformation drop-down list (e.g. Ecore to RDBMS) for future use. We note that toggling on semantics mode or `use CS` once a transformation is applied will have as effect to display the semantic state or the concrete syntax visualization of the transformed language model as defined by the target language.
- The displayed language model can be saved as XML or exported to XMI using the rightmost buttons ( ).
 - Language models are saved by default in the instance folder and following the format set by the Alloy analyzer. The consequence is that any instance obtained and saved from Lightning can be opened in the Alloy analyzer and vice-versa.
 - Language models can be exported to XMI, the default format in which Ecore models are stored. This functionality is experimental (not fully functional) and requires an Ecore translation of the language's *ASM*.

6.1.4 Instance Editor

To create/edit language models, Lightning offers to its users a very simple editor support. The instance editor is depicted in Fig. 6.3:

- On the left hand-side of this figure is the editor view that depicts in an editable tree the edited language model. Any non top-level node of the tree can be renamed and removed. The top level nodes represent signatures. Through the right-click context menu, it is possible to create an atom as a child of the given signature. Right-clicking an atom, allows to add tuples typed by fields declared in the signature typing the given atoms, e.g., in Fig. 6.3, adding a tuple of type `source` whose first atom is `Flow$0` is done by right clicking `Flow$0` and by selecting `source` in the context menu. The second atom composing the created tuple is to be selected in a drop-down list when creating the tuples.

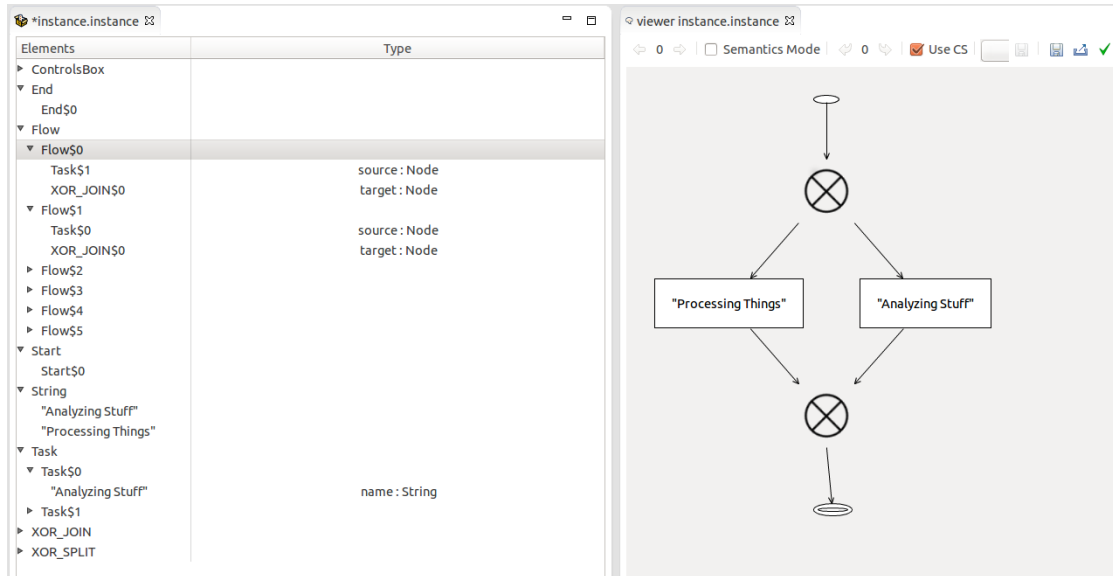


Figure 6.3: An overview of the Lightning Tree-based instance editor

- On the right hand-side, an instance viewer enables the visualization of the edited language model (using the concrete syntax definition if `use CS` is ticked). This instance viewer is paired to the tree editor so that any changes brought to the instance triggers an update of the instance viewer. This contributes to increasing the intuitiveness of the instance editor as users can directly measure the impact of their changes. Note as well that every change in the instance triggers a validation check consisting in verifying that the edited language model is indeed a valid instance of the language's abstract syntax model. A small flag (✓ or ✗) will let the user know whether or not the language model is a valid ASM-instance.

6.2 RPSL Meets Lightning, a Success Story

This section is dedicated to report a successful application of the Lightning tool in the frame of an interdisciplinary collaboration with Nico Hochgeschwender, at the time PhD candidate in robotic engineering at the University of Bonn in Germany. Nico is the creator of a language called RPSL allowing the specification of Robot Perception Systems. Lightning was used to formalize this language in Alloy and to perform so called “design space explorations”.

6.2.1 The RPSL Language

An inherent challenge to be faced when developing complex robotic systems – i.e., software systems deployed on robots – is the high variability [135, 136] of the environment in which the systems have to operate.

To deal with the complexity of considering multiple variabilities in the design of such system, it becomes more and more common for robotic engineers to adopt model driven approaches, one of them being the use of domain specific modeling languages.

One such language, named RPSL [137] (Robot Perception System Language), aims at considering two natures of variabilities affecting the design of robot perception systems, namely *functional and architectural variabilities*. In a nutshell, functional variability

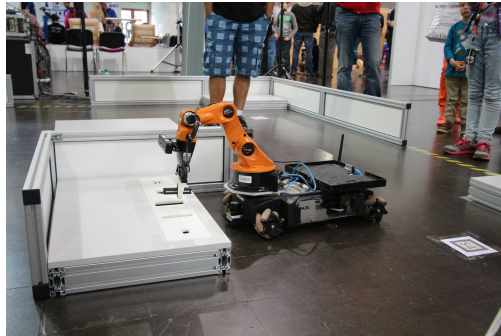


Figure 6.4: A youBot robot performing an insertion task at a service area.

refers to the vast amount of ways in which systems achieve their goal, each way being relevant in a fixed context, while architectural variability refers to the variation of goals themselves, robots operating in a dynamic environment.

To illustrate functional and architectural variability, let us consider the example of a robot performing an insertion task in an industrial environment (see Fig. 6.4). In order to perform this task, a robot perception system needs to, e.g., identify an object to grab for insertion.

An example of functional variability would then be to select the best strategy to identify the object. E.g., if the object is moving, object tracking should be used, else object detection is sufficient. An example of architectural variability would then be to propose different implementations of object detection based on different features of an object, be it color or shape.

In RPSL, functional variability is represented using feature models and architectural variability is represented using so called perception graphs (directed acyclic graph where nodes are sensor and processing components¹ and where two components are connected by an edge if the output of one can be used as the input of the other). The intent of RPSL is for each leaf feature representing a perception capability to be realized by one or several perception graphs. This feature to perception graph mapping is given in a so called *Resolution Model*.

RPSL specifications define what we call a *domain model*. A domain model is composed of a feature model, of a set of perceptual graphs and of a resolution model.

We then call *design alternative* a selection of features and perception graphs so that

1. the feature selection respects the dependencies declared in the feature model
2. exactly one perception graph is selected for each selected feature, respecting the mapping of the resolution model

For a given design alternative:

The configuration is the set of features and perception graphs selected.

The super graph is a well-formed composition of all the perception graphs in a configuration.

Relations between RPSL and design alternatives concepts are depicted in Fig. 6.5.

In short, a design alternative represents a possible implementation of the system defined by the domain model.

¹processing component require input while sensor only produces outputs

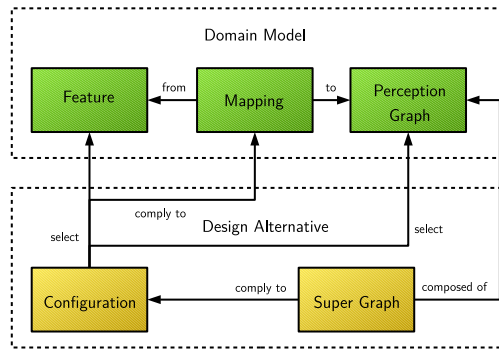


Figure 6.5: Structural overview of an RPSL domain model and its conforming design alternative.

Given an RPSL specification, it is necessary to ensure that all design alternatives conforming to a given domain model are relevant. The exercise of reviewing all design alternatives to identify those whose implementation are not desired is called *design space exploration*.

In the next section we illustrate the use of the RPSL language with a small case study.

6.2.2 The Pick & Place Case Study

The Pick & Place case study has been implemented using RPSL in the context of a recent robot competition, namely RoboCup@Work [138]. In this case study, a youBot mobile manipulation robot (see Fig. 6.4) is deployed in a factory-like environment which is composed of service areas. Each service area represents a region of the factory having a specific purpose for a particular task. For example, areas to load objects, to insert objects into object-specific cavities and to place objects into containers. Depending on a goal specification given by some factory worker the task of the robot is to pick objects such as screws, nuts and profiles from containers and to place and eventually insert them at corresponding service areas.

The functional variability of this scenario is given in RPSL in Listing 6.1 and depicted graphically in the upper-part of Fig. 6.6.

```

rpsl.feature_model do
  name "Pick and Place"
  add_feature "Application",      :is_root
  add_feature "ServiceArea",     :is_mandatory ,
                                :child_of = "Application"
  add_feature "ObDetection",     :child_of = "Application"
  add_feature "ObRecognition",   :requires = "ObDetection",
                                :child_of = "Application"

  add_feature "ContRecognition", :child_of = "Application"
  add_feature "CavRecognition",  :child_of = "Application"
end

```

Listing 6.1: Feature Model used in the Pick & Place case study specified in RPSL

The feature model contains five leaf features representing in the same order the following perceptual functionality required for the pick and place task:

- The service area detection feature allows to delimit the service area by detecting the dominant plane in the surroundings of the robot. This information is required by all other features as objects, container and cavities are all lying on this dominant plane.

```
rpsl.sensor_component do
  name "Kinect"
  add_port :out, "outCloud", "xyzRGB"
end

rpsl.processing_component do
  name "PlaneDetect"
  add_port :in, "inCloud", "xyzRGB"
  add_port :out, "outPlane", "Plane"
end

rpsl.perception_graph do
  name "Service Area 1"
  connect "Kinect", "outCloud", "PlaneDetect", "inCloud"
end
```

Listing 6.2: a Perception Graph implementing the Service Area feature expressed in RPSL

- The object detection feature provides a bounding box for each object present in the service area.
- The object recognition feature provides, when possible, a pose and a label for each detected object. Object detection is thus required by this feature.
- The container recognition feature provides, when possible, a pose and a bounding box for each container present in the service area.
- The cavity recognition feature provides, when possible, a pose for each cavity present in the service area.

The RPSL specification of a perception graph associated to the Service Area feature is given in Listing 6.2.

This perception graph specification proposes to perform the service area detection by using a kinect and a plane detection algorithm (e.g. RANSAC). Note that the kinect is declared as a sensor component whose output port is called outcloud (typed xyzRGB) and the plane detection algorithm is declared as a processing component whose input port is called inCloud (typed xyzRGB) and output port is called outPlane (typed Plane).

We note that this perception graph composes the super graph depicted in Fig. 6.6 as it is a possible implementation of the selected feature *ServiceArea*.

In the next section, we present how an RPSL design space exploration framework has been embedded in Lightning, before illustrating its use with the above case study.

6.2.3 A Lightning-Based Design Space Exploration Framework

The key to efficient design space exploration resides in (1) being able to derive the set of all design alternatives conforming to a given domain model and (2) in being able to review those design alternatives through an intuitive visualization. Lightning, allowing (1) the generation of instances using Alloy analysis and (2) the domain specific visualization of such instances, is thus a suitable environment in which to perform design space exploration.

A framework consisting of a set of models and scripts has hence been developed around Lightning to enable the seamless performance of design space exploration by RPSL domain experts. This framework consists of:

1. Alloy models formalizing:

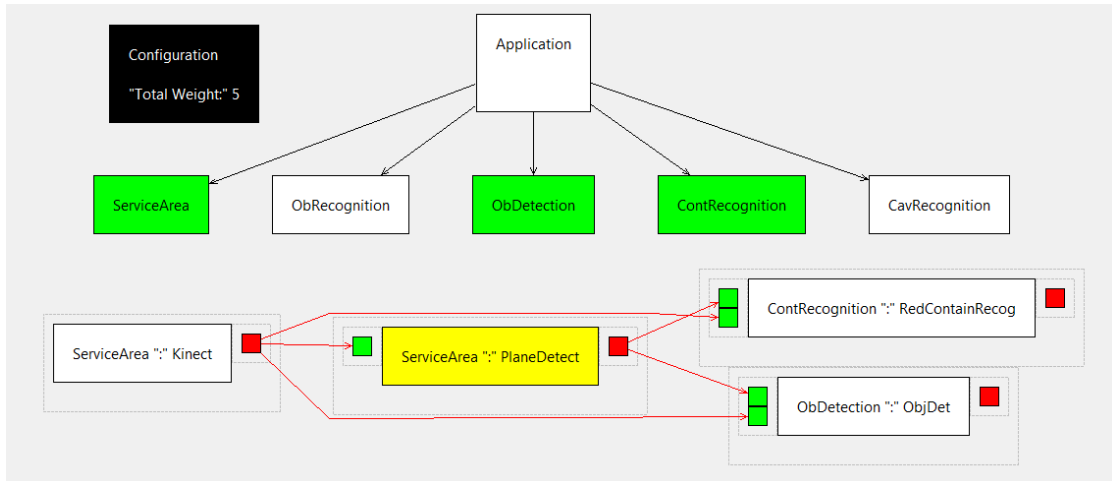


Figure 6.6: Visualization of a given Pick & Place configuration obtained with the framework. The feature model in which selected features are highlighted in green is at the top. At the bottom is a possible super graph for the given feature selection, with input and output ports displayed in green and red, respectively, and with components being either white or yellow depending on their associated weights.

- (a) RPSL's metamodel, by defining RPSL feature trees, perception graphs and resolution models.
 - (b) Design Alternatives, by modeling the concepts of configuration and super-graph reusing concepts of the RPSL metamodel.
2. F-Alloy transformations providing:
- (a) a domain specific visualization of RPSL specifications.
 - (b) a domain specific visualization of Design Alternatives.
3. a script translating RPSL textual specifications into Alloy models (instantiating RPSL's metamodel concepts).

This framework is depicted in Fig.6.7, each component being labeled after the indices given in the above enumeration.

In the following we give details on the implementation of some of those components.

Feature Tree Metamodel

The feature tree metamodel defines the set of valid feature models expressible in RPSL. In RPSL, a feature model is a tree of features where each child feature is a realization of its parent. Features can be mutually exclusive or require one another. The relationship between parent and children in an RPSL feature model are of two kinds, *specification* or *containment*. Their semantics differ when it comes to feature selection. When a parent feature is selected to be part of a design alternative, any children feature can be selected in the case of a containment relationship, while exactly one should be selected in the case of a specification relationship.

The Alloy model defining the feature tree metamodel is given in Listing 6.3.

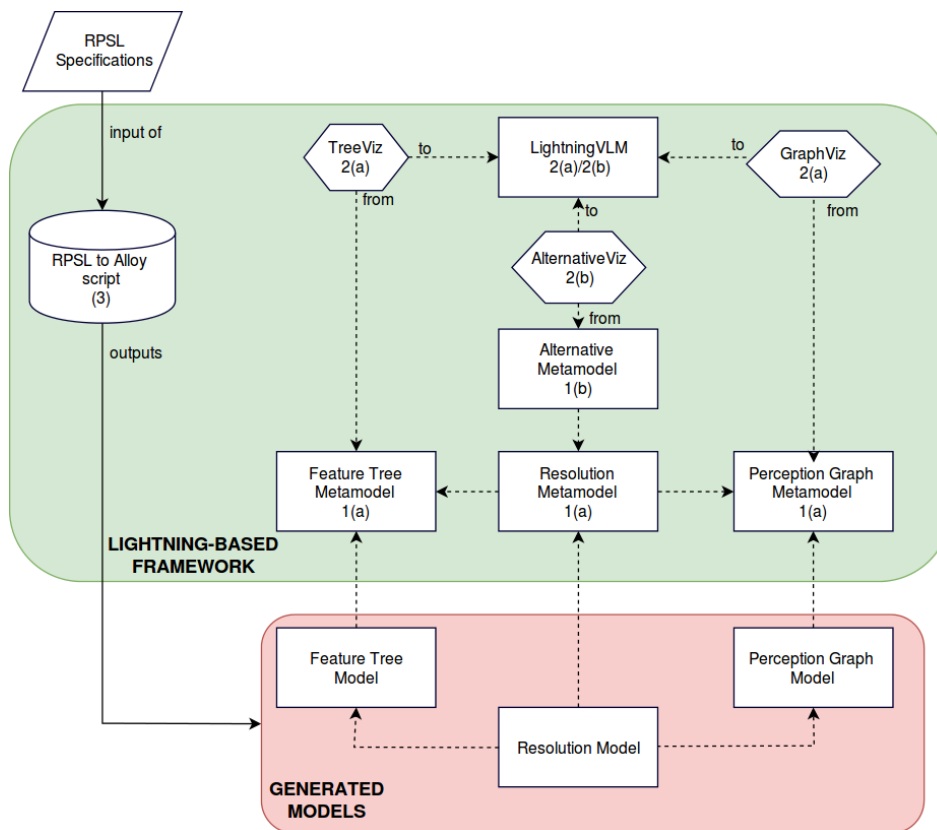


Figure 6.7: An overview of the Lightning-based Framework designed to facilitate the design space exploration of the RPSL specification. Note that boxes represent Alloy modules, hexagones represent f-modules and the dashed arrows represent import relations between modules.


```

module FeatureTreeMetamodel

abstract sig FeatureTree{
  root: Feature
}
{
  root.~(contain+spec)=none // no incoming edges
}

abstract sig Feature{
  spec: lone Feature,
  contain: set Feature,
  excluded: set Feature,
  required: set Feature
}
{
  some contain implies no spec
  some spec implies no contain
  let rel= (@spec+@contain){ // rel is the set of all edges composing the feature tree
    this not in this.^rel // prevent loops in the feature tree
    this.~*rel & required.@excluded =none // given a feature f, features required by f can't exclude
      f and f's parents
    one (this.*~rel & FeatureTree.root) // there's only one root feature in the parents of a given
      feature.
    lone (this.~@contain + this.~@spec )// any feature has at most one parent.
    excluded & required = none // a feature can't exclude and require the same feature

    excluded & this.*rel=none // a feature can't exclude itself or its children
    excluded & this.*~rel=none // a feature can't exclude itself or its parent
    required & this.*rel=none // a feature can't require itself or its children
    required & this.*~rel=none // a feature can't require itself or its parent
    no disj f1,f2:FeatureTree | this in f1.root.*rel and ((excluded+required) & f2.root.*rel) !=none
      // features of one feature tree can't exclude or require features of another feature tree.
    all x: excluded+required | some f:Feature | some disj f2,f3 :Feature | f2+f3 in f.@contain and
      (this+x) in (f2+f3).*rel and (this & f2.*rel) +(x & f3.*rel) =none // a feature can require or
      exclude another one only if both have ancestors which are (or are themselves) different
      alternatives of a same containment
    // for a given feature "this", let x be the set of excluded and required feature. We don't want
    another feature that is not an ancestor of this but which is a specialisation alternative of an
    ancestor of this and an ancestor of x.
    all x:excluded+required | no f:Feature | f not in this.*~rel and one f.~@spec and f.~@spec in
      this.*~rel and x in f.*rel // required and excluded features should be in the same
      specialisation branch.
  }
}

```

Listing 6.3: The RPSL feature tree metamodel expressed in Alloy

Perception Graph Metamodel

The perception graph metamodel defines the set of valid perception graphs expressible in RPSL. A perception graph in RPSL is a directed acyclic graph whose nodes represent sensor and processing components. Each component has output ports, while processing components also have input ports, both of arbitrary type¹. Input and output ports can be connected by an edge only if they share the same type. It is possible to define graphs containing solely one or several processing components in which case some input ports might not be connected. However, the composition of two (or more) such graphs is said to be *well-formed* if and only if all input ports are connected to some output ports. Each component can be assigned integer-valued weights as a discriminating factor with respect to a given property. The weight of a graph is the sum of weights of its components.

The Alloy module defining the Perception graph metamodel is given in Listing 6.4.

Resolution Metamodel

The resolution metamodel simply defines the concept of mapping between an RPSL feature and its implementing RPSL perception graphs. The resolution metamodel is given in Listing 6.5.

This resolution metamodel enables the generic definition of design alternatives (see below).

6.2.4 Alternatives Metamodel

The alternatives metamodel defines for given RPSL specification (generated by the RPSL to Alloy script) the set of valid design alternatives.

The alternatives metamodel is hence defined in terms of concepts defined in the feature tree, perception graph, and resolution metamodel. It thus imports the resolution metamodel (as shown in Figure 6.7). However we note that changing the import from the resolution metamodel to a resolution model corresponding to a given RPSL specification will enable, as we will see in section 6.2.5, the generation of design alternatives conforming to the said RPSL specification.

In Listing 6.6 we give the Alloy representation of the alternatives metamodel, defining the notions of configuration and super graph.

A domain specific visualization has been defined for each metamodel following the approach presented in Section 2.2.3. In Listing 6.7 we provide an excerpt of the F-Alloy transformation, named `AlternativeViz`, defining how instances of the alternatives metamodel have to be visualized.

6.2.5 A Design Space Exploration Scenario

An overview of how an RPSL domain expert can interact with the framework is given in Fig.6.8. In the following we illustrate each of those interactions through the help of the Pick & Place case study introduced in Section 6.2.2.

Step1: RPSL to Alloy

The first step toward the use of our design space exploration framework is the translation of the RPSL specifications to explore, like the ones given in Listing 6.1 and 6.2, into Alloy models.

¹We assume the existence of different types associated to each port. For more details about types we refer the reader to [137]

```

module PerceptionGraphMetamodel

abstract sig PerceptualGraph{
  components: set Component,
  connections : set Output -> Input,
  compGraph : set Component -> Component
}{
  // all Input and output in connections belongs to the components of the graph.
  all port:connections[Output] + connections.Input | port in components.(input+output)
  // connections between input and output only possible if type is the same
  all out : connections.Input | out.type = connections[out].type
  // ensure that compGraph ( a convenience field ) reflects the actual IO connections
  all disj c1,c2 : components| c2 in compGraph[c1] <=> c2 in connections[c1.output].~input
  no c:components| c in c.^compGraph // we don't want loops in our perceptual graph
  compGraph[Component]+ compGraph.Component in components // components in compgraph are actual
  components of the perceptual graph.
}

abstract sig Concept{}
abstract sig Port {
  type:Concept
}

abstract sig Input extends Port {}{
  this in Component.input
}

abstract sig Output extends Port {}{
  this in Component.output
}

abstract sig Component {
  input: disj set Input,
  output: disj set Output,
  weight: Int
}{
  weight in 1+2+3
  this in PerceptualGraph.components
}

abstract sig SensorComponent extends Component {}{
  #(input) = 0 and #(output) > 0
}

abstract sig ProcessingComponent extends Component {}{
  #(input) > 0 and #(output) > 0
}

// graph utility pred/functions
pred PerceptualGraph::contains(g: PerceptualGraph){
  g.components in this.components
  g.connections in this.connections
}

fun PerceptualGraph::getWeight():Int{
  sum c:this.components| c.weight
}

```

Listing 6.4: The RPSL Perception Graph metamodel expressed in Alloy

```

module ResolutionMetamodel

open FeatureTreeMetamodel
open PerceptionGraphMetamodel

one sig feature2Graph{
  mapping: Feature one -> some PerceptualGraph
}

```

Listing 6.5: The RPSL Resolution metamodel expressed in Alloy

```

module AlternativeMetamodel
open ResolutionMetamodel

one sig Configuration{
  selectedFeatures: set Feature,
  selectedGraph: set PerceptualGraph,
}{
  all f:selectedFeatures | one p:PerceptualGraph| f->p in feature2Graph.mapping and p in
    selectedGraph
  no disj x,y:selectedFeatures.~*(spec+contain) | x.excluded=y
  selectedFeatures.required in selectedFeatures.~*(spec+contain)
  selectedFeatures.(contain+spec)=none
}
one sig SuperGraph extends PerceptualGraph{
}{
  no c : components| c.input not in connections[Output]
  components=Configuration.selectedGraph.@components
  this.contains[Configuration.selectedGraph]
}

```

Listing 6.6: The RPSL alternatives metamodel expressed in Alloy

```

module AlternativeViz
open AlternativeMetamodel
open LightningVLM

one sig CREATE{
  mainFrame : Component -> INVISIBLE_CONTAINER,
  inputFrame: ProcessingComponent -> INVISIBLE_CONTAINER,
  component : Component -> RECTANGLE,
  inputPort: Input -> RECTANGLE,
  outputFrame: Component -> INVISIBLE_CONTAINER,
  outputPort: Output -> RECTANGLE,
  arc: Output -> Input -> CONNECTOR,
}
pred guard_component (c:Component) {
  c in SuperGraph.components
}
pred value_component (c:Component, r:RECTANGLE) {
  r.color=(c.weight=1 implies WHITE else (c.weight=2 implies YELLOW else ORANGE))
}
pred guard_arc(o:Output, i:Input) {
  o->i in SuperGraph.connections
}
pred value_arc(o:Output, i:Input, c:CONNECTOR) {
  c.source=CREATE.outputPort[o]
  c.target=CREATE.inputPort[i]
  c.color=RED
}

```

Listing 6.7: Excerpt of an F-Alloy transformation from the alternatives metamodel to the visual language metamodel defining how the supergraph is to be rendered

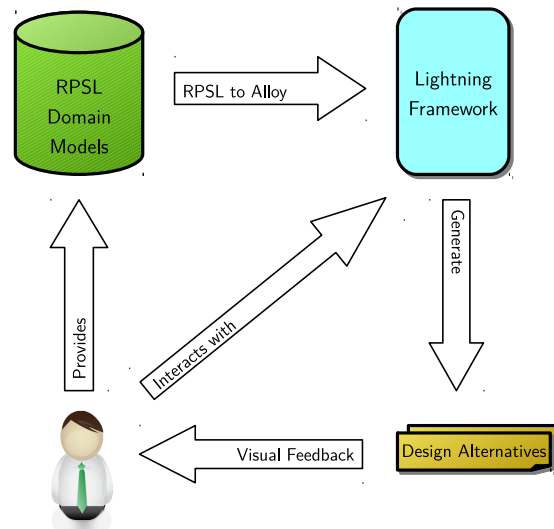


Figure 6.8: Overview of the usage of the Lightning-based framework in the performance of design space exploration. Here, a domain expert (bottom-left) can provide the framework with RPSL specifications, interact with the framework, and inspect graphically rendered design alternatives

The translation is done automatically as it is very straightforward: each element of the RPSL specification is declared as a singleton signature extending a type declared in of the relevant framework metamodel (e.g. a type declared in the `FeatureTreeMetamodel` if the translated RPSL specification is a feature model).

We illustrate this translation by giving in Listing 6.8 the Alloy model derived from the RPSL specification of the perception graph given in Listing 6.2.

```

module Service1Graph
open PerceptualGraphMetamodel

one sig Service1Graph extends PerceptualGraph{
  components = PlaneDetect + Kinect
  connections = outCloud -> inCloud
  compGraph = Kinect -> PlaneDetect
}

one sig inCloud extends Input {}{
  type=xyzRGB
}

one sig outPlane extends Output {}{
  type=Plane
}

one sig outCloud extends Output {}{
  type=xyzRGB
}

one sig Kinect extends SensorComponent{
  input = none and output = outCloud
}

one sig PlaneDetect extends ProcessingComponent {}{
  input = inCloud and output = outPlane
}
  
```

Listing 6.8: Alloy translation of the RPSL snippet given in Listing 6.2

Step 2: Generation and Visualization of Design Alternatives

Once Alloy modules – representing the feature tree, perception graph and resolution model of the RPSL specification to explore – are obtained as per Step 1, we can obtain

using Alloy analysis on the alternatives metamodel (e.g., Listing 6.6) the set of all design alternatives. To do so, the alternatives metamodel is automatically refined to import the Resolution Model obtained in Step 1 rather than the Resolution Metamodel (imported initially to express the Alternative Metamodel in terms of resolution, feature tree and perception graph model concepts).

Given the alternatives metamodel instances obtained by Alloy analysis and the `AlternativeViz` transformation defined in the framework (see Listing 6.7), domain specific visualizations of the design alternatives, such as the one given in Fig. 6.6, are returned to the domain expert.

The tree in the upper part of the visualization represents the feature tree of this case study, in which selected features are highlighted in green. For readability's sake the alternatives metamodel to Visual language transformation was modified to mask requirement arrows. The lower part of the visualization depicts the super graph resulting from the composition of perception graphs mapped in the resolution model to the highlighted selected features. Note that this super graph was not specified in RPSL and is resulting from the Alloy analysis of those well constrained models.

The red and green squares surrounding each component are their output and input ports, respectively. Note that, the `PlaneDetect` component appears in yellow as it is assigned a weight of 2 in the RPSL specifications. The black box in the top left corner lists additional properties of the selected configuration (here the total weight of the super graph implementing the features selected).

Step 3: Guiding the Design Space Exploration

Domain experts can further guide the exploration by defining additional constraints in the alternatives metamodel or by changing the weights assigned to each component. Adding constraints has as effect to reduce the number of possible instances of the Alternative model, thus narrowing the set of design alternatives to be considered. This mechanism becomes useful when the domain expert is interested in design alternatives showcasing specific properties. We list in the following some examples of constraints used to guide the design space exploration of our case study:

- **Specific Feature/Component Selection:** we were interested in reviewing all the design alternatives implementing the `ObRecognition` feature and whose supergraph contains a sensor providing `xyzRGB` data in order to ensure that `ObjRecognition` can be carried out for this kind of input data. The Alloy constraint used to express this is:

```
ObRecognition in Configuration.selectedFeatures and some (SensorComponent & xyzRGB.~type.~
output ) & SuperGraph.components
```

- **Optimal Solution Selection (with respect to the attributed weights):** we were interested in reviewing design alternatives with exactly three features implemented and having a minimal weight.

```
#Configuration.selectedFeatures = 3 and SuperGraph.getWeight[] < n
```

with `n` incrementally increasing until a design alternative is found.

6.3 Related Work on Language Workbenches

To understand what Lightning brings to the software language engineering scene, we compare its features to those of existing language workbenches. We base our comparison

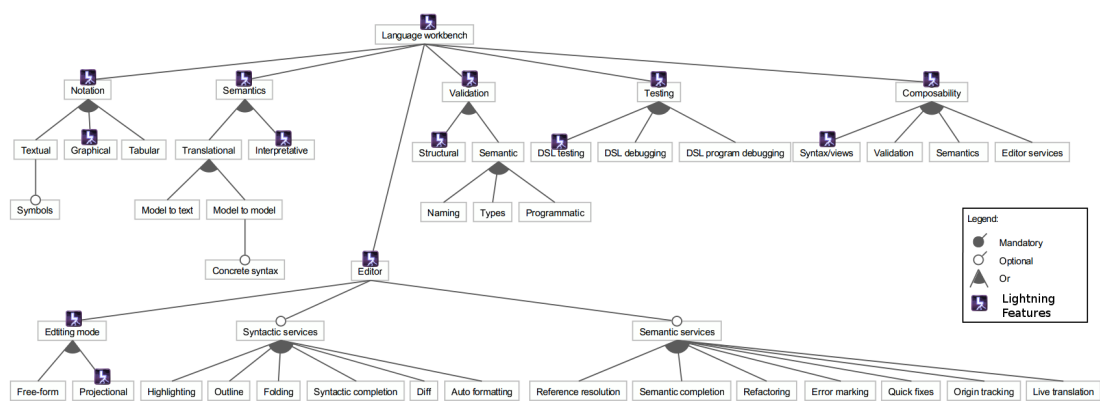


Figure 6.9: Generic Language Workbench Feature Model proposed in [5] with highlighting of the features covered by Lightning

on the domain analysis performed by [5] which resulted in the feature tree shown in fig. 6.9. We mark by a lightning symbol (⚡) all the features of this tree which are also provided by the Lightning tool and detail those features below.

Notation: The notation that Lightning uses for presenting models to the user is graphical: models are either shown in a concrete syntax notation (defined by a transformation of the abstract syntax model to a visual language) that is not editable, or in the form of a tree representation that can be edited.

Editor: The only editor support that Lightning currently offers is a tree based editor, supplemented by a graphical view of the model being edited (based on the concrete syntax of the language). Compared to mature language workbenches Lightning is clearly lacking comfortable editor support. This is shown in the lack of syntactic and semantic editor services. In contrast, the language workbenches MetaEdit+ [131], MPS [139], Spoofox [140], and Xtext [141] offer the full range of syntactic editor services – highlighting, outline, folding, syntactic completion, diff, and auto formatting. These same language workbenches also offer a range of semantic editor services such as reference resolution, semantic completion, refactoring and error marking.

While many existing workbenches offer advanced editor support, features that support validation and testing are less commonly found. Lightning offers at this point limited validation of models: upon saving a language model that does not conform to the language definition (i.e., the saved ASM-instance is not valid), an error is shown. On the other hand the MPS workbench, for instance, offers extended support for model validation, namely, structural, naming, types and programmatic validation.

Validation: The strength of Lightning lies in the validation mechanisms made available at the level of the language definition. The entry in the feature tree under DSL testing (see fig. 6.9) insufficiently reflects this support. At the level of the language definition Lightning offers advanced syntactic validation via Alloy. More importantly Alloy’s automatic analysis verifies the consistency of the language definition by generating sample instances. If no instances can be found, the abstract syntax definition is inconsistent. A similar validation can be carried out at the level of the semantics definition. For both syntax and semantics the visualization based on the concrete syntax definition aids in understanding the generated instances and the ensuing problems in the current specification.

Testing: Although some workbenches such as MPS offer support for testing at the level of the language definition, the SAT based analysis of Alloy is arguably more com-

plete since it exhaustively tests all instances up to a given size. Assuming that the small scope hypothesis which Alloy is based on - stating that errors usually admit small counterexamples - does indeed hold for language definitions, then the Lightning tool will indeed uncover problems in the language specification.

Composability: Regarding the last category of features, composability, Lightning at this point offers mostly syntactic composition via module import in Alloy. Other workbenches such as Metaedit+ and Spoofox offer a complete set of composability features covering all aspects of a language: syntax, validation, semantics, and editor services. In the eclipse world, the Melange language workbench enables to define new DSMLs as a composition of pre-existing ones. This is mainly achieved through the specification in the Melange DSL [142] of how Ecore models and Kermet3 aspects [143], defining the abstract syntax and semantics of each language, respectively, are to be merged [144].

Formal Methods: An aspect that is not covered in the feature tree given in Fig. 6.9 is whether or not the language workbench is based on formal methods in the sense that those play an integral part in the design process of the language. The formal underpinning of Lightning is a distinguishing characteristics. The workbenches reviewed in [5] do not have a direct formal foundation. The ATOM3 tool [130] (only mentioned in [5]) is the only formal language workbench presented as it relies mainly on graph transformations. Having formal methods involved in the design of DSLs at the earliest stage is a way to have increased confidence on the correctness of one's design.

6.4 Summary

In this Chapter, we have introduced Lightning, a tool enabling the design of DSMLs using Alloy and F-Alloy. We have seen that Lightning can be considered as a language workbench as it provides all the essential features (identified in [5]) of a language workbench. Lightning is to our knowledge one of the only language workbenches heavily focusing on design validation and natively supporting it. A current limitation of Lightning is its lack of integration with other works, mainly due to the choice of using Alloy to design languages. Despite this, we have proven the usefulness of Lightning by successfully designing RPSL and validating its design following the feedbacks of his creator. From this definition of RPSL, we successfully implemented a framework to so called design space exploration.

Chapter 7

Conclusion

In this concluding chapter, we provide a summary of our contributions in Section 7.1 and propose in Section 7.2 different research directions which can be followed to extend the work presented in this thesis.

7.1 Contributions

This thesis aimed at improving the current practices of Software Language Engineering with a particular focus on domain specific modeling languages and review-based validation. To reach this objective, we investigated the use of Alloy in the design of DSMLs. The use of the formal language Alloy in this exercise was motivated by (1) its high-level minimalist syntax allowing us to focus on system abstractions in a platform independent fashion and (2) by the ability of its accompanying tool, the Alloy analyzer, to generate instances from any Alloy specification, hence allowing seamless review-based validation. Those investigations had several outcomes, first of which is a novel approach to the definition of DSMLs based exclusively on Alloy. More precisely, we define how each component of a DSML definition (abstract syntax, concrete syntax and semantics) can be expressed by models and model transformations specified in Alloy, respectively. Based on this approach to DSMLs specification we introduced a design process tailored to enable the involvement of domain experts in their validation.

The Lightning language workbench, a tool implementing the aforementioned approach and allowing the performance of its associated design process, is introduced. The process of computing model transformations using Alloy analysis being quite time consuming, the Lightning language workbench in its first release suffered great usability issues.

To tackle those limitations, we identified a subset of the Alloy language having the property of being interpretable. From this subset we created a new model transformation language called F-Alloy.

We also coined the notion of hybrid analysis, an analysis process relying both on Alloy analysis and F-Alloy interpretation to efficiently generate instances from F-Alloy specifications. We have shown that hybrid analysis enables to reduce the analysis time of any (possibly compound) transformation to that of the Alloy analysis of its (leftmost) source module. Based on this hybrid analysis, we introduced a novel approach to the validation of model transformations named VBV (Visualization Based Validation), consisting in generating domain specific traces of execution from the transformation specification to validate.

Finally we have presented a real world experience from using Lightning in the definition and validation of RPSL (a robotic DSML) specifications. This experience comforts

the claim that our contributions are of significance to those aspiring to formally specify and validate their DSMLs.

7.2 Future Work

Limited by time, several research tracks related to the use of Alloy in the context of domain specific modeling languages engineering remain to be explored. In this section, we list the two tracks we believe are the most relevant.

7.2.1 The F-Alloy Track

The F-Alloy language is the end-result of our investigations in the domain of model transformations. The scope of these investigations was, in the frame of this thesis, limited to the textual specifications of functions. This scope could be broadened in the future as suggested in the following.

Graphical Concrete Syntax: One of the potential limitations of F-Alloy is the prerequisite of being familiar with Alloy in order to use the language. A possible future work could thus be to provide an alternative graphical syntax to F-Alloy to make the exercise of specifying model transformations accessible to domain experts. A work in progress attempting to represent F-Alloy specifications graphically is provided in [145]. Following this line of work, a graphical concrete syntax for F-Alloy could ultimately be defined so as to allow the automatic visualization of F-Alloy specifications.

Bidirectional Model Transformations: Another area of investigation concerns bidirectional transformations. These are transformations that allow forward and backward transformations to be generated from a unique transformation specification. Bidirectional transformations are useful in the context of synchronization between models. In the context of our work, there is a need for F-Alloy to allow the specification of bidirectional transformation. Indeed, being able to synchronize abstract and concrete syntax will enable Lightning to provide a more advanced language model editor support, as it would then be possible to seamlessly reverberate any changes brought to the concrete syntax (through editing) back to the abstract syntax. Future work should thus determine if the F-Alloy language can be adapted to enable the specification of efficiently computable bidirectional transformations.

Expressiveness: Finally, despite having an handful of academic case studies already implemented using F-Alloy¹, a more thorough evaluation of the language's expressiveness could be provided, possibly by implementing industrial cases studies.

7.2.2 The Lightning Track

Lightning is the embodiment of our advancements in the domain of Alloy-based DSML engineering. Reviewing missing Lightning features is thus a way to identify relevant future work related to that domain.

Full Editor Support: We have seen in Chapter 6 that the editor support provided by Lightning is limited to a tree editor whose structure depends on the abstract syntax. This limitation is due to the concrete syntaxes being defined in Lightning through

¹<http://lightning.gforge.uni.lu/examples>

a unidirectional F-Alloy transformation (from the abstract syntax to a VLM). Consequently synchronization between abstract syntax and concrete syntax is non trivial. A master thesis [146] aimed at identifying ways of synchronizing abstract and concrete syntax (other than defining the concrete syntax as a bidirectional transformation). The objectives of this master thesis were to (1) determine the possibility of deriving a partial definition of a backward transformation (from VLM to ASM) from a concrete syntax defined using Lightning (as an F-Alloy transformation from ASM to VLM), to (2) identify the information missing in the concrete syntax definition to derive the complete corresponding backward transformation and (3) to implement a web editor deployable from a Lightning language definition, allowing the edition of language model through their concrete syntax (using the derived backward transformation). It turned out that the 3rd objective of implementing an easily deployable web-based language model editor was a non-trivial task that required the full thesis time to be completed, hence leaving open the research questions addressed by the thesis. Future work should thus focus on bringing answers to those questions. It is to be noted that once synchronization is achieved, such editor would natively support advanced features such as validation and model completion. Validation could be performed on the fly by evaluating facts declared in the language's abstract syntax model. Generation of intuitive error messages when a fact is violated could be achieved by simply annotating each fact of the abstract syntax. The auto-completion feature (building on demand a valid language model from an invalid one by applying the least amount of changes) could also easily be implemented by relying on KodKod [147, 148], the relational model finder Alloy is based on.

Collaboration Support While Lightning enables domain experts and language engineers to work hand in hand in the specification of DSMLs, only limited collaboration features are available – i.e., those provided by existing Eclipse plug-ins [149]. The lack of an unified platform in which all actors of the DSML design process could communicate and work on the DSML design concurrently prevents fluid remote collaborations. A recently developed platform called Collaboro [150] shows the benefits of enabling a collaborative design process through powerful collaborative features like live communication, vote-based decision, a.s.o... One can imagine that a collaborative design process could also be supported by Lightning the moment the aforementioned online editor deployment feature is implemented. Indeed, the deployed online editor could be used by domain experts to test language designs and publish instant feedback to the attention of language engineers working on eclipse. Such feedback mechanism could easily rely on an existing ticketing system, already well integrated to Eclipse. The collaborative design process would then be composed of short test-fix iterations.

Integration With Other Existing Tools: The Lightning language workbench's main focus is on validation of DSML designs, somehow to the detriment of other activities revolving around DSMLs like textual concrete syntax definition [151, 152], DSML composition [153, 154], code generation [155, 156], collaborative design [157, 158], and so on. Instead of implementing home-brewed features enabling those activities to be undertaken in Lightning, it would be interesting to investigate ways of connecting Lightning to already existing solutions. Lightning has been released as an Eclipse plug-in having this work direction in mind. The main challenge of connecting Lightning to other solutions is to translate Alloy models and instances produced using Lightning to Ecore models and XMI instances (and vice versa). While this challenge is already addressed by works in [77, 78], we believe that another interesting problem to consider is the investigation of ways to integrate those external tools into Lightning and its agile design process capability. As a concrete example, let us consider the problem of defining textual concrete syntaxes. The nowadays leading eclipse-based tool dedicated to this purpose

is Xtext [141]. Currently, to validate a DSL designed in Xtext, one has to compile the Xtext specifications to generate the editor as an Eclipse plug-in project, then launch the generated eclipse project, and finally start fiddling with the editor to see if syntax validation indeed works as expected. Such a heavy validation process could greatly benefit from the more lightweight analysis based validation provided by Lightning, which could consist in simply taking as input the Xtext specification and returning examples of syntactically valid/invalid code for the purpose of validating them.

Annexe A

Functional Alloy Module Expressing the CD2RDBMS Transformation

```

module CD2RDBMS
open CD
open RDBMS

one sig CREATE{
  class2table: Class -> Table,
  attribute2column: Attribute -> Column,
  association2column: Association -> Attribute -> Column,
  association2FKey: Association -> FKey,
}

pred guard_class2table(c:Class){
  c.parent=none
}
pred value_class2table(c:Class , t:Table){
  t.name[0]=c.name
}

pred guard_attribute2column(a:Attribute){
  a not in AssociationClass.attributes
}
pred value_attribute2column(a:Attribute , c:Column){
  c.dataType=(a.type.name="String" implies "TEXT" else "NUMBER")
  c.name[0]= a.name
  c.name[1]=((a.~attrs.parent)!=none implies a.~attrs.name else none)
  all i:Int| i>=1 and i< #(a.~attrs.^parent) implies c.name[add[i,1]]= c.name[i].~name.parent.name
  a.is_primary=True implies c in CREATE.class2table[a.~attrs.*parent].pkeys
  c in CREATE.class2table[a.~attrs.*parent].cols
}

pred guard_association2column(ass:Association,att:Attribute){
  att.is_primary=True and att in ass.dest.attrs
}
pred value_association2column(ass:Association , att:Attribute, c:Column){
  c.dataType=(att.type.name="String" implies "TEXT" else "NUMBER")
  c.name[0]=ass.name
  c.name[1]=att.name
  c in CREATE.class2table[ass.src].cols
}

pred guard_association2FKey(a:Association){}
pred value_association2FKey(a:Association , f:FKey){
  f.references=CREATE.class2table[a.dest]
  f.columns=CREATE.association2column[a,Attribute]
  f in CREATE.class2table[a.src].fkeys
}

fact C_PRE_C_POST{
  all x : Class {
    (guard_class2table[x] and one CREATE.class2table[x] and value_class2table[x, CREATE.
      class2table[x]])
  }
}

```

```

    or
    (not guard_class2table[x] and no CREATE.class2table[x])
  }

  all x : Attribute{
    (guard_attribute2column[x] and one CREATE.attribute2column[x] and value_attribute2column[x,
      CREATE.attribute2column[x]])
    or
    (not guard_attribute2column[x] and no CREATE.attribute2column[x])
  }

  all x : Association {
    (guard_association2FKey[x] and one CREATE.association2FKey[x] and value_association2FKey[x,
      CREATE.association2FKey[x] ])
    or
    (not guard_association2FKey[x] and no CREATE.association2FKey[x] )
  }
  all x:Association,x1:Attribute{
    (guard_association2column[x,x1] and one CREATE.association2column[x,x1] and
      value_association2column[x,x1, CREATE.association2column[x,x1]])
    or
    (not guard_association2column[x,x1] and no CREATE.association2column[x,x1])
  }
}

fact C_TRACE_EX{
  RDBMSElement = CREATE.class2table[Class] + CREATE.attribute2column[Attribute]+ CREATE.
    association2column[Association,Attribute]
  FKey=CREATE.association2FKey[Association]
}

```

Annexe B

Functional Alloy Module Expressing the CDRefinement Transformation

```
module CDRefinement
open CD

one sig CREATE{
  associationClass2Class: AssociationClass -> Class,
  newAssociations: Class-> Association -> Association
}

pred guard_associationClass2Class(a:AssociationClass){}
pred value_associationClass2Class(a:AssociationClass,y:Class){
  y.name= a.association.name
  y.attrs=a.attributes
  y.is_abstract=False
  y.parent=none
}

pred guard_newAssociations(c:Class,a:Association){
  c in a.(src+dest) and a.~association!none
}
pred value_newAssociations(c:Class,a:Association,y:Association){
  y.name= a.name+c.name
  y.src=(c=a.src implies c else CREATE.associationClass2Class[a.~association])
  y.dest=(c=a.dest implies c else CREATE.associationClass2Class[a.~association])
}

one sig UPDATE{
  fixAbstract: Class -> Class
}

pred guard_fixAbstract(c:Class){
  c.is_abstract=True and c.~parent=none
}
pred value_fixAbstract(c:Class,y:Class){
  y.is_abstract=False
  y.attrs=c.attrs
  y.name=c.name
  y.parent=c.parent
}

one sig DELETE{
  associationWithClass: set Association,
  associationClass : set AssociationClass
}

pred guard_associationWithClass(a:Association){
  a.~association!none
}
pred guard_associationClass(a:AssociationClass){
}

fact C_PRE_C_POST{
```

```

    all x : AssociationClass {
        (guard_associationClass2Class[x] and one CREATE.associationClass2Class[x] and
         value_associationClass2Class[x, CREATE.associationClass2Class[x]])
    or
        (not guard_associationClass2Class[x] and no CREATE.associationClass2Class[x])
    }
    all x : Class |all y: Association{
        (guard_newAssociations[x,y] and one CREATE.newAssociations[x,y] and value_newAssociations[x
        ,y, CREATE.newAssociations[x,y]])
    or
        (not guard_newAssociations[x,y] and no CREATE.newAssociations[x,y])
    }
    all x : Class{
        (guard_fixAbstract[x] and one UPDATE.fixAbstract[x] and value_fixAbstract[x, UPDATE.
        fixAbstract[x]])
    or
        (not guard_fixAbstract[x] and no UPDATE.fixAbstract[x])
    }
    all x : Association {
        (guard_associationWithClass[x] and x in DELETE.associationWithClass)
    or
        (not guard_associationWithClass[x] and x not in DELETE.associationWithClass)
    }
    all x : AssociationClass {
        (guard_associationClass[x] and x in DELETE.associationClass)
    or
        (not guard_associationClass[x] and x not in DELETE.associationClass)
    }
}

fact C_TRACE_EN_1{
    no CREATE.associationClass2Class[AssociationClass] & CREATE.newAssociations.Association.
    Association
    no CREATE.associationClass2Class[AssociationClass] & UPDATE.fixAbstract.Class
    no UPDATE.fixAbstract[Class] & CREATE.newAssociations.Association.Association
    no UPDATE.fixAbstract[Class] & UPDATE.fixAbstract.Class
    no CREATE.newAssociations[Class , Association] & CREATE.newAssociations[Class].Association
    no CREATE.newAssociations[Class , Association] & DELETE.associationWithClass
}

fact C_TRACE_EN_2{
    let input = univ - (CREATE + DELETE + UPDATE + UPDATE.fixAbstract[Class] + CREATE.
    associationClass2Class[AssociationClass] + CREATE.newAssociations[Class, Association]){
    let output = univ - (CREATE + DELETE + UPDATE + DELETE.(associationWithClass +
    associationClass) + UPDATE.fixAbstract.Class) {
        attrDisj[input] and attrDisj[output]
        acyclicInheritance[input] and acyclicInheritance[output]
        nameDisj[input] and nameDisj[output]
        noOrphanAttr[input] and noOrphanAttr[output]
        AssociationFact[input] and AssociationFact[output]
    }
}
}
}

```


ATL Implementation of the CD2RDBMS Transformation

```

1 module cd2rdbms;
2 create OUT : RDBMS from IN : CD;
3
4 rule PersistentClass2Table{
5   from
6     c : CD!Class (
7       c.is_persistent and c.parent->oclIsUndefined()
8     )
9   using ...
10  to
11    t : RDBMS!Table (
12      name<-c.name,
13      cols<-primary_key_columns->union(foreign_key_columns)->union(rest),
14      pkeys<-primary_key_columns,
15      fkeys<-foreign_keys
16    ),
17
18    primary_key_columns : distinct RDBMS!Column foreach (primAttr in primary_attributes)
19      (
20        name<-primAttr.name,
21        dataType<-primAttr.type.name
22      ),
23
24    foreign_keys : distinct RDBMS!FKey foreach (persAttr in persistent_features)
25      (
26        references<-persAttr.class.topParent,
27        columns<-persistent_features->iterate(tuple;
28          acc : Sequence(Sequence(RDBMS!Column))=Sequence{} |
29          acc->append(foreign_key_columns.subSequence(
30            tuple.offcet,
31            tuple.offcet + tuple.noFAttrs-1)))
32      ),
33    foreign_key_columns : distinct RDBMS!Column foreach (attr in foreign_key_attributes)
34      (
35        name<-attr.name,
36        dataType<-attr.type.name
37      ),
38    rest : distinct RDBMS!Column foreach (attr in rest_of_attributes)
39      (
40        name<-attr.name,
41        dataType<-attr.type.name
42      )
43 }

```

Listing C.1: a CD2RDBMS transformation expressed in ATL(adapted from [159])

This transformation is composed of a single rule named `PersistentClass2Table` (line 4). The rule is executed for each topmost persistent class `c` (line 7). This rule produces a table (line 11) named after class `c` (line 12), containing a set of column (line 13), some of which are primary keys (line 14), foreign keys (line 15) or neither. The set of primary key columns produced from a class `c` is defined in lines 18-22. The set of foreign keys produced from a class `c` is defined in lines 24-32, and the columns these contain in lines 33-37. Columns that are neither primary nor foreign keys are obtained from line 38-42.

Annexe D

A Visual Language Model Expressed in Alloy: the LightningVLM

```

module LightningVLM
open util/ternary
open util/boolean
abstract sig Layout{}
one sig VERTICAL_LAYOUT extends Layout{}
one sig HORIZONTAL_LAYOUT extends Layout{}
abstract sig VisualElement{
    color : Lightning_Color
}
abstract sig Symbol extends VisualElement{
    composedOf: seq VisualElement,
    layout:Layout
}
abstract sig Shape extends Symbol{}
sig INVISIBLE_CONTAINER extends Shape{}
sig RECTANGLE extends Shape{}
sig TRIANGLE extends Shape{}
sig ELLIPSE extends Shape{}
sig RHOMBUS extends Shape{}
sig CYLINDER extends Shape{}
sig ACTOR extends Shape{}
sig CLOUD extends Shape{}
sig HEXAGON extends Shape{}
sig DOUBLE_ELLIPSE extends Shape{}
sig IMAGE extends Symbol{
    url: String
}
sig TEXT extends VisualElement {
    textLabel: seq univ,
    isBold : Bool,
    isItalic :Bool,
}{ #textLabel=#(textLabel.elems)}
fact compositionSanity{
    all s : Symbol | s not in s.^(composedOf.select13)
    all ve: VisualElement| #(ve.~(composedOf.select13))<2
}
abstract sig Lightning_Color{}
one sig RED extends Lightning_Color{}
one sig GREEN extends Lightning_Color{}
one sig BLUE extends Lightning_Color{}
one sig ORANGE extends Lightning_Color{}
one sig YELLOW extends Lightning_Color{}
one sig PURPLE extends Lightning_Color{}
one sig BROWN extends Lightning_Color{}
one sig BLACK extends Lightning_Color{}
one sig GRAY extends Lightning_Color{}
one sig WHITE extends Lightning_Color{}

```

Listing D.1: A Visual Language Model defined in Alloy. Instances of this model can be parsed by the Lightning tool and rendered accordingly



Glossary

Abstract Syntax A part of a DSL definition defining the set of valid language models. 12

Abstract Syntax Model (ASM) A representation of the abstract syntax as an Alloy module. 86

Alloy A lightweight formal language allowing the declarative specification of systems in term of concepts, relations and constraints. 2

Alloy Expression There are two main categories of expressions in Alloy: Boolean-valued (used to define constraints) and set-valued (yielding a set of atoms/tuples). 16

Assertion A checkable Alloy expression. Checking it leads to the generation of counter-examples. 15

Command An instruction defining what kind of analysis is to be performed on the module it is declared in as well as within what scope. 15

Fact An constraint expressed in Alloy that should always be satisfied . 15

Field The declaration of a relation between concepts in Alloy. 15

Function A parametrized, set-valued, Alloy expression . 15

Let instruction assigning an expression to a variable than can either be used locally (inside a block) or globally (throughout the module) . 15

Predicate A parametrized, boolean-valued, Alloy expression . 15

Scope Scopes define the size of instances that can be generated from an Alloy module. Concretely, they are upper-bound to the number of atoms typed by each signature. 13

Signature The declaration of a concept in Alloy. 15

Alloy Instance A generated model conforming to the metamodel specified by an Alloy module. 15

Atom Representation of an object taking its type from a signature . 15

Conforming A model is said to conform to a metamodel if it follows the structure and satisfies the constraints defined by it. 3

Counter Example Obtaining from checking an assertion, counter-examples are instances (satisfying all invariants) in which the checked assertion is violated. . 15

Tuple Representation of a relation between objects taking its type from a field. 15

Analysis A process allowing to obtain conforming models (or counter-examples) from a given metamodel. 2

Alloy Analysis Analysis performed by the Alloy Analyzer on given Alloy modules. It relies on SAT-solving to generate instances and counter-examples. 17

Concrete Syntax A part of a DSL definition defining how to represent language models. 12

CRUD is an acronym used to refer to the four basic operations performable on persistent storages, namely: Create, Read, Update, Delete. This term is also used in the model transformation community when describing endogenous in-place operations . 11

Domain Expert Given a system, the domain expert is familiar with its domain of application, but has no knowledge of the technology used to design it. 2

Domain Specific Language (DSL) A language specifically designed to solve a limited set of problems. They are to put in contrast with general purpose languages like Java, which have a wider expressiveness. 12

Domain-specific modeling languages (DSMLs) A DSL whose specific aim is to model systems. 2, 12

Engineer We call engineer someone who has technical knowledge. 1

Language Engineer An engineer specialized in the definition of DSLs. 12

Transformation Engineer An engineer specialized in model transformation technologies. 72

F-Alloy A sublanguage of Alloy allowing the specification of efficiently computable functions. 3

Guard Predicate A predicate specifying the condition under which source and target elements are to be part of a mapping. 30

Mappings Backbones of an F-Alloy transformation, they define relations between source and target module. 18

Rule An F-Alloy expression having the property of being interpretable . 37

Traceability Links Tuples typed after mappings . 19

Value Predicate A predicate, containing rules, specifying what must hold for elements that are part of a mapping . 30

Function A non-deterministic single-output model transformation. 24

Language Model an occurrence of the language. Compared to a model, it is understood that a language model can be viewed and executed using the language's well defined concrete syntax and semantics, respectively. 12

Language Workbench A tool specialized in the definition and usage of DSLs. 13

Lightning An Alloy-based language workbench prototype, proof of concept of this thesis' work. 4

LightningVLM The default Visual Language Metamodel provided by Lightning. 99

Model A conceptual representation of chosen aspects of the system in which details irrelevant to those aspects are abstracted away. 1

Model Transformation Executable specifications enabling to automatically modify or create from a model, various artifacts, from model to code..

compound A transformation whose source or target metamodels are transformation specifications. 3

Endogenous transformations whose source and target metamodels are the same. 3, 10

Exogenous transformations whose source metamodel differ from their target metamodel. 3, 10

Higher Order A transformation whose source or target metamodels are themselves transformation languages. 11

In-place the transformation defines how to modify the source model in order to obtain the target model. 3, 11

Out-place the transformation defines how to build from scratch the target model from the source model. 11

Module A file containing a metamodel specification expressed in the Alloy language. 14

Alloy Module metamodel specification expressed in Alloy. 14

F-Module function specification expressed in F-Alloy. 36

Functional Alloy Module function specification expressed in Alloy. 25

Source Module source of an F-Alloy transformation . 18

Target Module target of an F-Alloy transformation . 18

Semantics A part of a DSL definition providing meaning to language models. 12

Denotational Associate mathematical objects to language models. 13

Operational Define how language models are to be executed as a sequence of semantics steps. 13

Pragmatic Language model's meaning are defined by the execution of a tool processing them. 13

Translational Enables the translation of a language model into another language whose semantics is well defined. 13

UML The Unified Modeling Language emerged from an effort of standardizing model representations. It comprises a set of standard languages allowing to model every aspects of a system.. 95

Validation the exercise of ensuring that a design represents effectively the desired system. 1

Verification the exercise of ensuring that a design is correct, that is, it is devoid of design errors. 1

Visualization Based Validation (VBV) A validation process relying on graphically reviewing instances . 4

VLM A Visual Language Metamodel aims at defining a graphical language in terms of concepts and relations. . 12

Bibliography

- [1] X. Thirioux, B. Combemale, X. Crégut, and P.-L. Garoche, “A framework to formalise the MDE foundations,” in *Proceedings of the International Workshop on Towers of Models (TOWERS 2007)*, pp. 14–30, University of York, 2007.
- [2] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, “Model transformations? Transformation models!,” in *Proceedings of the MoDELS conference*, vol. 4199, pp. 440–453, Springer, 2006.
- [3] E. Syriani, *A multi-paradigm foundation for model transformation language engineering*. PhD thesis, McGill University, 2011.
- [4] F. Hermann, H. Ehrig, F. Orejas, and U. Golas, “Formal analysis of functional behaviour for model transformations based on triple graph grammars,” in *Proceedings of the International Conference on Graph Transformation*, pp. 155–170, Springer, 2010.
- [5] S. et al. Erdweg, “The state of the art in language workbenches,” in *Proceedings of the International Conference on Software Language Engineering (M. Erwig, R. F. Paige, and E. Wyk, eds.)*, Lecture Notes in Computer Science, pp. 197–217, Springer International Publishing, 2013.
- [6] P. Naur and B. Randell, “Software engineering: Report on a conference sponsored by the NATO science committee,” Nato, 1969.
- [7] E. W. Dijkstra, “The humble programmer,” *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.
- [8] R. R. Schaller, “Moore’s law: past, present and future,” *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.
- [9] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, “A comparison of empirical and model-driven optimization,” *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 63–76, 2003.
- [10] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, “Combined iterative and model-driven optimization in an automatic parallelization framework,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2010.
- [11] F. Fleurey, J. Steel, and B. Baudry, “Validation in model-driven engineering: testing model transformations,” in *Proceeding of the 1st International Workshop on Model Design and Validation*, pp. 29–40, IEEE, 2004.
- [12] G. Holzmann and R. Joshi, “Model-driven software verification,” *Model Checking Software*, pp. 76–91, 2004.

- [13] F. Rekik, B. Bannour, S. Dhouib, and S. Gérard, “Model-driven consistency verification for service-oriented applications,” in *Proceedings of the 8th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 180–187, IEEE, 2015.
- [14] B. Graaf and A. van Deursen, “Model-driven consistency checking of behavioural specifications,” in *Proceedings of the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, pp. 115–126, IEEE, 2007.
- [15] M. L. Griss, “Software reuse architecture, process, and organization for business success,” in *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pp. 86–89, IEEE, 1997.
- [16] K. C. Kang, J. Lee, and P. Donohoe, “Feature-oriented product line engineering,” *IEEE software*, vol. 19, no. 4, pp. 58–65, 2002.
- [17] D. S. Kolovos, R. F. Paige, and F. A. Polack, “Scalability: The holy grail of model driven engineering,” in *Proceedings of the International Workshop on Challenges in Model-Driven Software Engineering (ChaMDE)*, pp. 10–14, 2008.
- [18] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, *et al.*, “A research roadmap towards achieving scalability in model driven engineering,” in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, p. 2, ACM, 2013.
- [19] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [20] A. Knapp and S. Merz, “Model checking and code generation for UML state machines and collaborations,” *Proceedings of the 5th Workshop on Tools for System Design and Verification*, pp. 59–64, 2002.
- [21] A. M. Davis, *Software requirements: analysis and specification*. Prentice Hall Press, 1990.
- [22] D. C. Schmidt, “Model-driven engineering,” *IEEE COMPUTER SOCIETY*, vol. 39, no. 2, p. 25, 2006.
- [23] A. R. da Silva, “Model-driven engineering: A survey supported by the unified conceptual model,” *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.
- [24] D. Jackson, *Software abstractions*. MIT Press Cambridge, 2012.
- [25] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani, “MDE adoption in industry: challenges and success criteria,” in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pp. 54–59, Springer, 2008.
- [26] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of MDE in industry,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 471–480, IEEE, 2011.

-
- [27] A. Iwai, N. Oohashi, and S. Kelly, “Experiences with automotive service modeling,” in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, p. 1, ACM, 2010.
- [28] Y. Ridene, N. Belloir, F. Barbier, and N. Couture, “A DSML for mobile phone applications testing,” in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, p. 3, ACM, 2010.
- [29] J.-P. Tolvanen and S. Kelly, “Defining domain-specific modeling languages to automate product derivation: Collected experiences,” *Software Product Lines*, pp. 198–209, 2005.
- [30] K. S. Barber, T. Graser, and J. Holt, “Providing early feedback in the development cycle through automated application of model checking to software architectures,” in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 341–345, IEEE, 2001.
- [31] A. Hegedus, G. Bergmann, I. Ráth, and D. Varró, “Back-annotation of simulation traces with change-driven model transformations,” in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 145–155, IEEE, 2010.
- [32] B. Combemale, L. Gonnord, and V. Rusu, “A generic tool for tracing executions back to a DSML’s operational semantics.,” *ECMFA*, vol. 6698, pp. 35–51, 2011.
- [33] K. Anastasakis, *A model driven approach for the automated analysis of UML class diagrams*. University of Birmingham, 2009.
- [34] T. T. Dinh-Trong, S. Ghosh, and R. B. France, “A systematic approach to generate inputs to test UML design models,” in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pp. 95–104, IEEE, 2006.
- [35] G. Georg, J. Bieman, and R. B. France, “Using Alloy and UML/OCL to specify runtime configuration management: A case study.,” in *pUML*, pp. 128–141, Citeseer, 2001.
- [36] M. Taghdiri and D. Jackson, “A lightweight formal analysis of a multicast key management scheme,” in *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems*, pp. 240–256, Springer, 2003.
- [37] R. M. Moreira and A. C. Paiva, “A novel approach using Alloy in domain-specific language engineering,” in *Proceedings of the Model-Driven Engineering and Software Development Conference (MODELSWARD)*, pp. 157–164, IEEE, 2015.
- [38] C. Huang, Y. Kamei, K. Yamashita, and N. Ubayashi, “Using Alloy to support feature-based DSL construction for mining software repositories,” in *Proceedings of the International Software Product Line Conference co-located workshops*, pp. 86–89, ACM, 2013.
- [39] J. J. López-Fernández, J. de Lara, and E. Guerra, *An agile process for the example-driven development of modelling languages and environments*. PhD thesis, Universidad Autónoma de Madrid, 2017.
- [40] H. Cho, Y. Sun, J. Gray, and J. White, “Key challenges for modeling language creation by demonstration,” in *Proceedings of the Workshop on Flexible Modeling Tools*, 2011.

- [41] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
- [42] K. Anastasakis, B. Bordbar, and J. M. Küster, “Analysis of model transformations via Alloy,” in *Proceedings of the 4th MoDeVVA workshop: Model-Driven Engineering, Verification, and Validation*, pp. 47–56, 2007.
- [43] N. Macedo and A. Cunha, “Implementing QVT-R bidirectional model transformations using Alloy,” in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pp. 297–311, Springer, 2013.
- [44] L. Gammaitoni and P. Kelsen, “Domain-specific visualization of Alloy instances,” in *Proceedings of the International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 324–327, 2014.
- [45] L. Gammaitoni, P. Kelsen, and F. Mathey, “Verifying modelling languages using Lightning: a case study,” *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA)*, pp. 19–28, 2014.
- [46] L. Gammaitoni and P. Kelsen, “F-Alloy: An Alloy based model transformation language,” in *Proceedings of the International Conference on Theory and Practice of Model Transformations*, pp. 166–180, Springer, Cham, 2015.
- [47] L. Gammaitoni, P. Kelsen, and C. Glodt, “Designing languages using Lightning,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 77–82, ACM, 2015.
- [48] L. Gammaitoni, P. Kelsen, and Q. Ma, “Agile validation of higher order transformations using F-Alloy,” in *Proceedings of the International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 125–131, IEEE, 2016.
- [49] L. Gammaitoni and N. Hochgeschwender, “RPSL meets Lightning: A model-based approach to design space exploration of robot perception systems,” in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAN)*, pp. 75–82, IEEE, 2016.
- [50] L. Gammaitoni, P. Kelsen, and Q. Ma, “Agile validation of model transformations using compound F-Alloy specifications,” *Science of Computer Programming journal*, 2017.
- [51] M. Tauber, D. Ackermann, and G. Fischer, “The importance of models in making complex systems comprehensible,” 1991.
- [52] F. Kordon, J. Hugues, and X. Renault, “From model driven engineering to verification driven engineering,” in *Proceedings of the International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pp. 381–393, Springer, 2008.
- [53] D. Moody, “What makes a good diagram? improving the cognitive effectiveness of diagrams in IS development,” in *Advances in Information Systems Development*, pp. 481–492, 2007.
- [54] P. Mohagheghi and V. Dehlen, *Where Is the Proof? A Review of Experiences from Applying MDE in Industry*, pp. 432–443. 2008.

-
- [55] J. Miller, J. Mukerji, M. Belaunde, *et al.*, “MDA guide,” *Object Management Group*, 2003.
- [56] “Common facilities RFP-5: Meta-Object Facility, cf/96-05-02,” June 1996.
- [57] T. Mens and P. Van Gorp, “A taxonomy of model transformation,” *Electronic Notes in Theoretical Computer Science*, pp. 125–142, 2006.
- [58] M. Tisi, S. Martínez, F. Jouault, and J. Cabot, “Refining models with rule-based model transformations,” 2011.
- [59] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, “On the use of higher-order model transformations,” *ECMDA-FA*, vol. 9, pp. 18–33, 2009.
- [60] P. Stevens, “A landscape of bidirectional model transformations,” *GTTSE*, pp. 408–424, 2007.
- [61] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.
- [62] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.
- [63] P. Hudak, “Domain-specific languages,” *Handbook of Programming Languages*, vol. 3, no. 39-60, p. 21, 1997.
- [64] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [65] M. Fowler, “Language workbenches: The killer-app for domain specific languages,” <http://martinfowler.com/articles/languageWorkbench.html>, 2005.
- [66] L. Baresi and P. Spoletini, “On the use of Alloy to analyze graph transformation systems,” in *Graph Transformations*, pp. 306–320, 2006.
- [67] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “On challenges of model transformation from UML to Alloy,” *Software & Systems Modeling*, pp. 69–86, 2010.
- [68] N. Macedo, T. Guimarães, and A. Cunha, “Model repair and transformation with Echo,” in *ASE’13*, pp. 694–697, IEEE, 2013.
- [69] A. Hall, “Realising the benefits of formal methods,” *J. UCS*, vol. 13, no. 5, pp. 669–678, 2007.
- [70] C. Bolton, “Using the Alloy analyzer to verify data refinement in Z,” *Electronic Notes in Theoretical Computer Science*, vol. 137, no. 2, pp. 23–44, 2005.
- [71] C. Pons and D. Garcia, “A lightweight approach for the semantic validation of model refinements,” *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 43–61, 2008.
- [72] M. R. Prasad, A. Biere, and A. Gupta, “A survey of recent advances in SAT-based formal verification,” *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.
- [73] F. Jouault and I. Kurtev, *Transforming Models with ATL*. 2006.

- [74] A. Schürr, “Specification of graph translators with triple graph grammars,” in *Graph-Theoretic Concepts in Computer Science*, pp. 151–163, 1995.
- [75] OMG, “Meta Object Facility Query/View/Transformation specification,” 2011.
- [76] M. Balaban, P. Bennett, K.-H. Doan, G. Georg, M. Gogolla, I. Khitron, and M. Kifer, “A comparison of textual modeling languages: OCL, Alloy, FOML,” in *OCL@ MoDELS*, pp. 57–72, 2016.
- [77] A. Cunha, A. Garis, and D. Riesco, “Translating between Alloy specifications and UML class diagrams annotated with OCL,” *Software & Systems Modeling*, vol. 14, no. 1, pp. 5–25, 2015.
- [78] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “UML2Alloy: A challenging model transformation,” in *Model Driven Engineering Languages and Systems*, pp. 436–450, 2007.
- [79] Y. He, “Comparison of the modeling languages Alloy and UML,” *Software Engineering Research and Practice*, vol. 2, pp. 671–677, 2006.
- [80] A. Schürr, “Specification of graph translators with triple graph grammars,” in *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, Springer, 1995.
- [81] K. Lano, “Catalogue of model transformations,” <http://dcs.kcl.ac.uk/staff/kcl/tcat.pdf>, 2014.
- [82] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, “Henshin: advanced concepts and tools for in-place EMF model transformations,” in *Model Driven Engineering Languages and Systems*, pp. 121–135, 2010.
- [83] D. H. Akehurst, S. Kent, and O. Patrascoiu, “A relational approach to defining and implementing transformations between metamodels,” *Software and System Modeling*, pp. 215–239, 2003.
- [84] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, “Transformation: The missing link of MDA,” in *Graph Transformation*, pp. 90–105, 2002.
- [85] K. Czarnecki and S. Helsen, “Classification of model transformation approaches,” in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pp. 1–17, 2003.
- [86] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *Science of computer programming*, pp. 31–39, 2008.
- [87] J. Troya and A. Vallecillo, “A rewriting logic semantics for ATL.,” *Journal of Object Technology*, pp. 1–29, 2011.
- [88] A. Schürr and F. Klar, “15 years of triple graph grammars,” in *Graph Transformations*, pp. 411–425, 2008.
- [89] H. Giese, S. Hildebrandt, and L. Lambers, “Toward bridging the gap between formal semantics and implementation of triple graph grammars,” in *Proceedings of the 7th MoDeVVA Workshop: Model-Driven Engineering, Verification, and Validation*, pp. 19–24, 2010.

-
- [90] J. Zhang, Y. Liu, J. Sun, J. S. Dong, and J. Sun, "Model checking software architecture design," in *Proceedings of the International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 193–200, IEEE, 2012.
- [91] S. Mesli-Kesraoui, D. Kesraoui, F. Oquendo, A. Bignon, A. Toguyeni, and P. Berruet, "Formal verification of software-intensive systems architectures described with piping and instrumentation diagrams," in *Proceedings of the European Conference on Software Architecture (ECSA)*, pp. 210–226, Springer, 2016.
- [92] A. Lin, M. Bond, and J. Clulow, "Modeling partial attacks with Alloy," in *Proceedings of the International Workshop on Security Protocols*, pp. 20–33, Springer, 2007.
- [93] P. S. Grisham, C. L. Chen, S. Khurshid, and D. E. Perry, "Validation of a security model with the Alloy analyzer," 2006.
- [94] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*, pp. 18–33, 2009.
- [95] K. Lano, T. Clark, and S. Kolahdouz-Rahimi, "A framework for model transformation verification," *Formal Aspects of Computing*, vol. 27, no. 1, pp. 193–235, 2015.
- [96] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," in *In Popl*, vol. 2, 2003.
- [97] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard, "Optimizations for compiling declarative models into boolean formulas," in *Theory and Applications of Satisfiability Testing*, pp. 187–202, 2005.
- [98] E. Uzuncaova and S. Khurshid, "Constraint prioritization for efficient analysis of declarative models," in *FM: Formal Methods*, pp. 310–325, 2008.
- [99] S. Ganov, S. Khurshid, and D. E. Perry, "Annotations for Alloy: automated incremental analysis using domain specific solvers," in *Formal Methods and Software Engineering*, pp. 414–429, 2012.
- [100] A. A. El Ghazi and M. Taghdiri, "Analyzing Alloy constraints using an SMT solver: a case study," in *Proceedings of the 5th International Workshop on Automated Formal Methods (AFM)*, 2010.
- [101] G. Henry, A. Bonneau, and V. Colotte, "Tools devoted to the acquisition of the prosody of a foreign language," in *Proceedings of the International Congress of Phonetic Sciences*, pp. 1593–1596, 2007.
- [102] J. Lönnberg, A. Korhonen, and L. Malmi, "MVT: a system for visual testing of software," in *Proceedings of the Working conference on Advanced visual interfaces*, pp. 385–388, ACM, 2004.
- [103] C. M. Park, S. M. Bajimaya, S. C. Park, G. N. Wang, J. G. Kwak, and K. H. Han, "Development of virtual simulator for visual validation of PLC program," in *Proceedings of the International conference on computational Intelligence for Modelling, Control and Automation*, pp. 32–32, IEEE, 2006.

- [104] K. Al-Kodmany, "Using visualization techniques for enhancing public participation in planning and design: process, implementation, and evaluation," *Landscape and urban planning*, vol. 45, no. 1, pp. 37–45, 1999.
- [105] S. Khurshid and D. Jackson, "Exploring the design of an intentional naming scheme with an automatic constraint analyzer," in *Proceedings of the International Conference on Automated Software Engineering, (ASE)*, pp. 13–22, 2000.
- [106] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 133–142, 2004.
- [107] N. Asoudeh and R. Khosravi, "Alloy as a language for domain modeling," *School of Electrical and Computer Engineering Teheran*, 2007.
- [108] A. Zaman, I. Kazerani, M. Patki, B. Guntoori, and D. Rayside, "Improved visualization of relational logic models," *University of Waterloo, Tech. Rep*, 2013.
- [109] M. F. van Amstel, M. G. J. van den Brand, and A. Serebrenik, "Traceability visualization in model transformations with TraceVis," in *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT)*, pp. 152–159, 2012.
- [110] A. Holl and G. Valentin, "Structured business process modeling (SBPM)," *Information systems research in Scandinavia (IRIS)*, 2004.
- [111] S. C. Tosatto, G. Governatori, and P. Kelsen, "Towards an abstract framework for compliance," in *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pp. 79–88, Los Alamitos, CA, USA: IEEE Computer Society, 2013.
- [112] Z. Demirezen, M. Mernik, J. Gray, and B. Bryant, "Verification of DSMLs using graph transformation: a case study with Alloy," in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, p. 3, ACM, 2009.
- [113] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.
- [114] OMG, "Unified modeling language (OMG UML)," *Superstructure*, 2007.
- [115] O. OMG, "2.0 specification," *Object Management Group, Final Adopted Specification*, 2005.
- [116] M. Richters, "The USE tool: A UML-based specification environment," *Internet: <http://www.db.informatik.uni-bremen.de/projects/USE>*, vol. 20, pp. 133–147, 2001.
- [117] M. Gogolla, "Towards model validation and verification with SAT techniques," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.
- [118] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software & Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

-
- [119] H. Eichelberger, Y. Eldogan, and K. Schmid, “A comprehensive survey of UML tool capabilities and compliance,” *University of Hildesheim, Institut für Informatik, Hildesheimer Informatik-Berichte, Tech. Report*, pp. 08–01, 2008.
- [120] G. Giachetti, B. Marín, and O. Pastor, “Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles,” in *Proceedings on the International Conference on Advanced Information Systems Engineering*, pp. 110–124, Springer, 2009.
- [121] T. Wiman, “UML-based domain-specific modeling using MetaModelAgent,” *Internet: <http://www.adocus.com/media/2566/UML-Based-Domain-Specific-Modeling-using-MetaModelAgent.pdf>*, 2010.
- [122] J. J. López-Fernández, E. Guerra, and J. de Lara, “Meta-model validation and verification with MetaBest,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 831–834, ACM, 2014.
- [123] Z. Yang and M. Jiang, “Using eclipse as a tool-integration platform for software development,” *IEEE Software*, vol. 24, no. 2, 2007.
- [124] J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara, “Example-driven meta-model development,” *Software & Systems Modeling*, vol. 14, no. 4, pp. 1323–1347, 2015.
- [125] J. J. López-Fernández, E. Guerra, and J. de Lara, “Combining unit and specification-based testing for meta-model validation and verification,” *Information Systems*, vol. 62, pp. 104–135, 2016.
- [126] J. J. López-Fernández, E. Guerra, and J. de Lara, “Assessing the quality of meta-models,” in *MoDeVVA@ MoDELS*, pp. 3–12, 2014.
- [127] J. J. López-Fernández, E. Guerra, and J. de Lara, “Example-based validation of domain-specific visual languages,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 101–112, ACM, 2015.
- [128] W.-M. Ho, F. Pennaneac’h, and N. Plouzeau, “UMLAUT: A framework for weaving UML-based aspect-oriented designs,” in *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages*, pp. 324–334, IEEE, 2000.
- [129] J. Deantoni, F. Mallet, and C. André, “On the formal execution of UML and DSL models,” in *WIP of the 4th International School on Model-Driven Development for Distributed, Realtime, Embedded Systems*, 2009.
- [130] J. De Lara and H. Vangheluwe, “AToM3: A tool for multi-formalism and meta-modelling,” in *Fundamental approaches to software engineering*, pp. 174–188, 2002.
- [131] S. Kelly, K. Lyytinen, and M. Rossi, “MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment,” in *Advanced Information Systems Engineering*, pp. 1–21, 1996.
- [132] J. Davis, “GME: the generic modeling environment,” in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 82–83, ACM, 2003.

- [133] Honeywell, “DoME guide,” <http://www.htc.honeywell.com/dome/>, 1992.
- [134] R. Bardohl, “GenGE: A generic graphical editor for visual languages based on algebraic graph grammars,” in *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, pp. 48–55, IEEE, 1998.
- [135] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, “Design abstraction and processes in robotics: From code-driven to model-driven engineering,” in *Proceedings of the 2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAN)*, (Darmstadt, Germany), 2010.
- [136] L. Gherardi and D. Brugali, “Modeling and reusing robotic software architectures: the HyperFlex toolchain,” in *IEEE International Conference on Robotics and Automation*, 2014.
- [137] N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar, “Declarative specification of robot perception architectures,” in *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAN)*, 2014.
- [138] G. K. Kraetzschmar, N. Hochgeschwender, W. Nowak, F. Hegger, S. Schneider, R. Dwiputra, J. Berghofer, and R. Bischoff, “RoboCup@Work: Competing for the Factory of the Future,” in *Proceedings of the 18th RoboCup International Symposium*, 2014.
- [139] M. Voelter and V. Pech, “Language modularity with the MPS language workbench,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 1449–1450, 2012.
- [140] L. C. L. Kats and E. Visser, “The Spoofox language workbench: rules for declarative specification of languages and IDEs,” in *ACM Sigplan Notices*, pp. 444–463, 2010.
- [141] M. Eysholdt and H. Behrens, “XText: implement your language faster than the quick and dirty way,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309, 2010.
- [142] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, “Melange: A meta-language for modular and reusable development of DSLs,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pp. 25–36, 2015.
- [143] J.-M. Jézéquel, O. Barais, and F. Fleurey, “Model driven language engineering with Kermeta,” in *Generative and Transformational Techniques in Software Engineering*, pp. 201–221, Springer, 2011.
- [144] T. Degueule, “Interoperability and composition of DSLs with Melange,” 2016.
- [145] L. Gammaitoni, “A Graphical Syntax for F-Alloy,” Tech. Rep. TR-LASSY-17-02, University of Luxembourg, 2017. made available at <https://lightning.gforge.uni.lu/doc/TR-LASSY-17-02.pdf>.
- [146] C. Kamphaus, “A web based graphical environment for using domain specific languages in Lightning,” Tech. Rep. TR-LASSY-17-01, University of Luxembourg, 2017. made available at <https://lightning.gforge.uni.lu/doc/TR-LASSY-17-01.pdf>.

-
- [147] E. Torlak and G. Dennis, “Kodkod for alloy users,” in *First ACM Alloy Workshop, Portland, Oregon*, 2006.
- [148] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 632–647, Springer, 2007.
- [149] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson, “Jazzing up eclipse with collaborative tools,” in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pp. 45–49, ACM, 2003.
- [150] J. L. C. Izquierdo and J. Cabot, “Collaboro: A collaborative (meta) modeling tool,” *PeerJ Computer Science*, vol. 2, p. e84, 2016.
- [151] F. Jouault, J. Bézivin, and I. Kurtev, “TCS: a DSL for the specification of textual concrete syntaxes in model engineering,” in *Proceedings of the 5th international conference on Generative programming and component engineering*, pp. 249–254, ACM, 2006.
- [152] H. Krahn, B. Rumpe, and S. Völkel, “Integrated definition of abstract and concrete syntax for textual languages,” in *MODELS*, vol. 4735, pp. 286–300, Springer, 2007.
- [153] A. Vallecillo, “On the combination of domain specific modeling languages,” *ECMFA*, vol. 10, pp. 305–320, 2010.
- [154] B. Combemale, *Towards Language-Oriented Modeling*. PhD thesis, Université de Rennes 1, 2015.
- [155] H. Krahn, B. Rumpe, and S. Völkel, “Roles in software development using domain specific modeling languages,” *arXiv preprint arXiv:1409.6618*, 2014.
- [156] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, “AtomPM: A web-based modeling environment,” in *Joint proceedings of MODELS’13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pp. 21–25, 2013.
- [157] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Leventovszky, and Á. Lédeczi, “Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure,” *MPM@ MODELS*, vol. 1237, pp. 41–60, 2014.
- [158] J. L. C. Izquierdo and J. Cabot, “Enabling the collaborative definition of DSMLs,” in *Proceedings of the International Conference on Advanced Information Systems Engineering*, pp. 272–287, Springer, 2013.
- [159] M. Igamberdiev, G. Grossmann, and M. Stumptner, “Verification of the CD2RDBMS transformation case in Flora-2: VOLT 2015 case study technical report,” tech. rep., Tech. rep., Knowledge and Software Engineering Lab, University of South Australia, 2015.